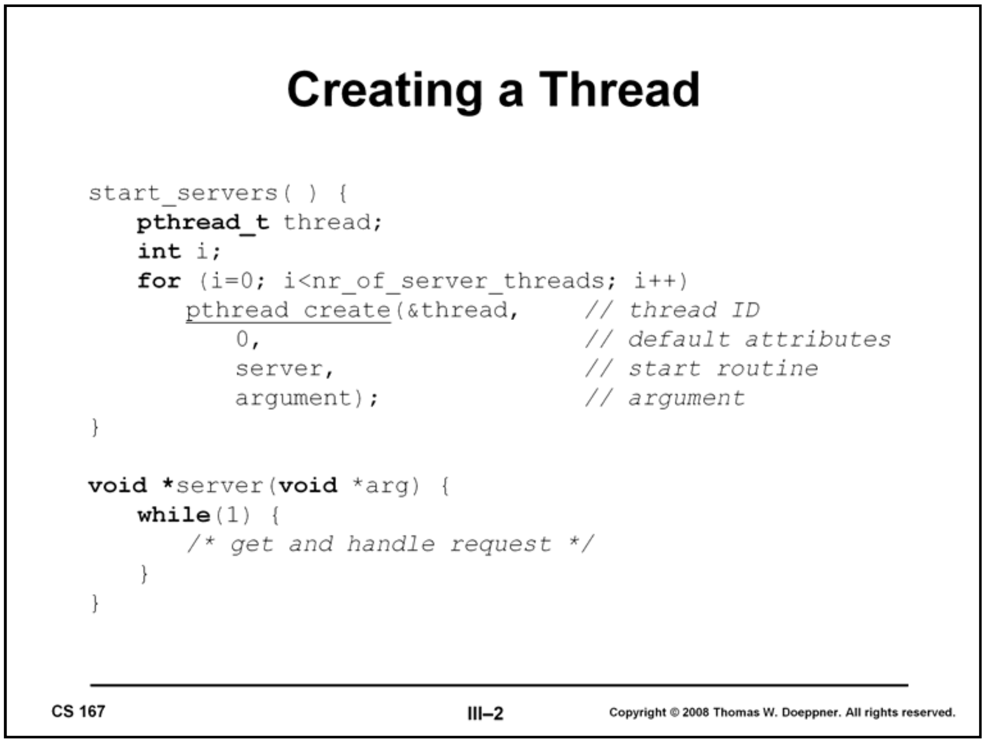
Programming with POSIX Threads I

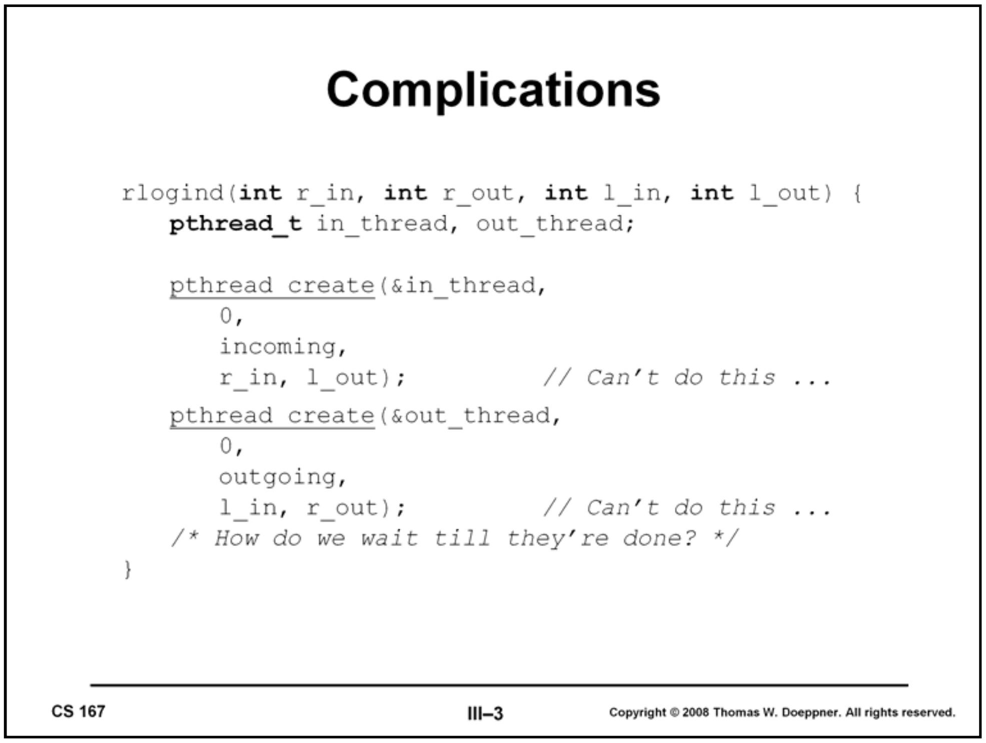


Creating a Thread

```
start_servers( ) {
   pthread_t thread;
   int i;
   for (i=0; i<nr_of_server_threads; i++)
      pthread_create(&thread,     // thread ID
         0,                       // default attributes
         server,                  // start routine
         argument);               // argument
}

void *server(void *arg) {
   while(1) {
      /* get and handle request */
   }
}
```

To create a thread, one calls the *pthread_create* routine. This skeleton code for a server application creates a number of threads, each to handle client requests. If *pthread_create* returns successfully (i.e., returns 0), then a new thread has been created that is now executing independently of the caller. This new thread has an ID that is returned via the first parameter. The second parameter is a pointer to an attributes structure that defines various properties of the thread. Usually we can get by with the default properties, which we specify by supplying a null pointer (we discuss this in more detail later). The third parameter is the address of the routine in which our new thread should start its execution. The last argument is the argument that is actually passed to the first procedure of the thread.

If *pthread_create* fails, it returns a code indicating the cause of the failure.

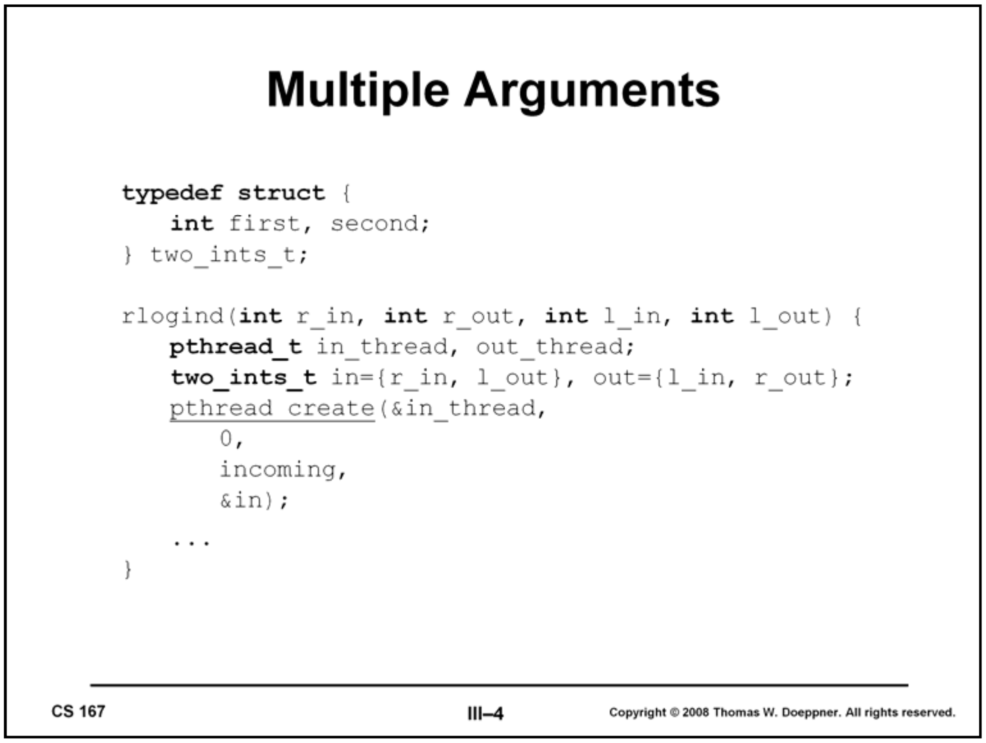


Complications

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
   pthread_t in_thread, out_thread;

   pthread_create(&in_thread,
      0,
      incoming,
      r_in, l_out);          // Can't do this ...
   pthread_create(&out_thread,
      0,
      outgoing,
      l_in, r_out);          // Can't do this ...
   /* How do we wait till they're done? */
}
```

An obvious limitation of the *pthread_create* interface is that one can pass only a single argument to the first procedure of the new thread. In this example, we are trying to supply code for the rlogind example of the previous module, but we run into a problem when we try to pass two parameters to each of the two threads.

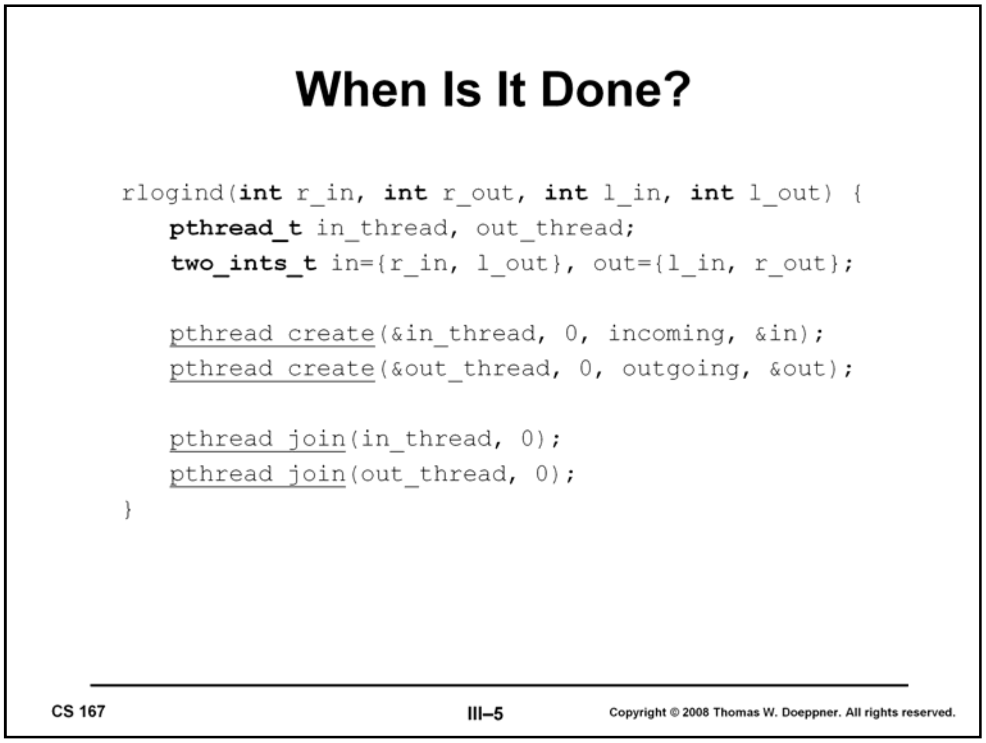
A further issue is synchronization with the termination of a thread. For a number of reasons we'll find it necessary for a thread to be able to suspend its execution until another thread has terminated.



Multiple Arguments

```
typedef struct {
    int first, second;
} two_ints_t;

rlogind(int r_in, int r_out, int l_in, int l_out) {
    pthread_t in_thread, out_thread;
    two_ints_t in={r_in, l_out}, out={l_in, r_out};
    pthread_create(&in_thread,
        0,
        incoming,
        &in);
    ...
}
```

To pass more than one argument to the first procedure of a thread, we must somehow encode multiple arguments as one. Here we pack two arguments into a structure, then pass the pointer to the structure. This technique at least does not produce any compile-time errors, but it has a potential serious problem.

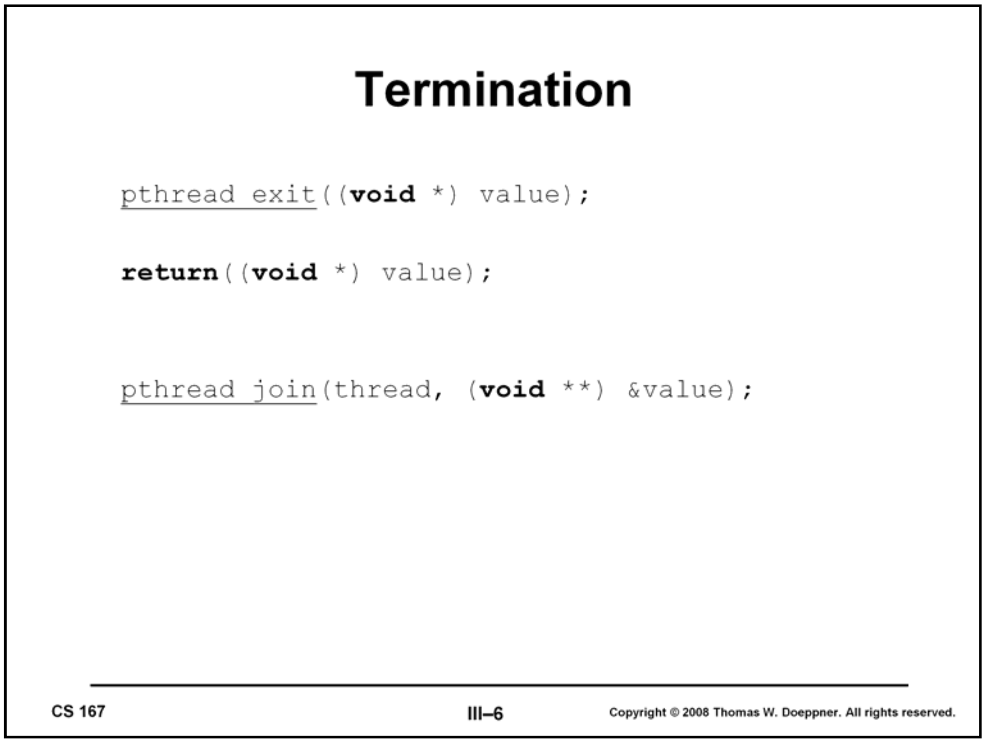


When Is It Done?

```
rlogind(int r_in, int r_out, int l_in, int l_out) {
   pthread_t in_thread, out_thread;
   two_ints_t in={r_in, l_out}, out={l_in, r_out};

   pthread_create(&in_thread, 0, incoming, &in);
   pthread_create(&out_thread, 0, outgoing, &out);

   pthread_join(in_thread, 0);
   pthread_join(out_thread, 0);
}
```

In the previous example, the *in* and *out* structures are local variables of the “mainline” thread. Their addresses are passed to new threads. If the mainline thread returns from its rlogind procedure before the new threads terminate, there is the danger that the new threads may dereference their argument pointers into storage locations that are no longer active. This would cause a serious runtime problem that might be difficult to track down.

However, if we can guarantee that the mainline thread does not return from rlogind until after the new threads have terminated, then our means for passing multiple arguments is safe. In the example above, the mainline thread places calls to *pthread_join*, which does not return until the thread mentioned as its first argument has terminated. Thus the mainline thread waits until the new threads terminate and then returns from rlogind.



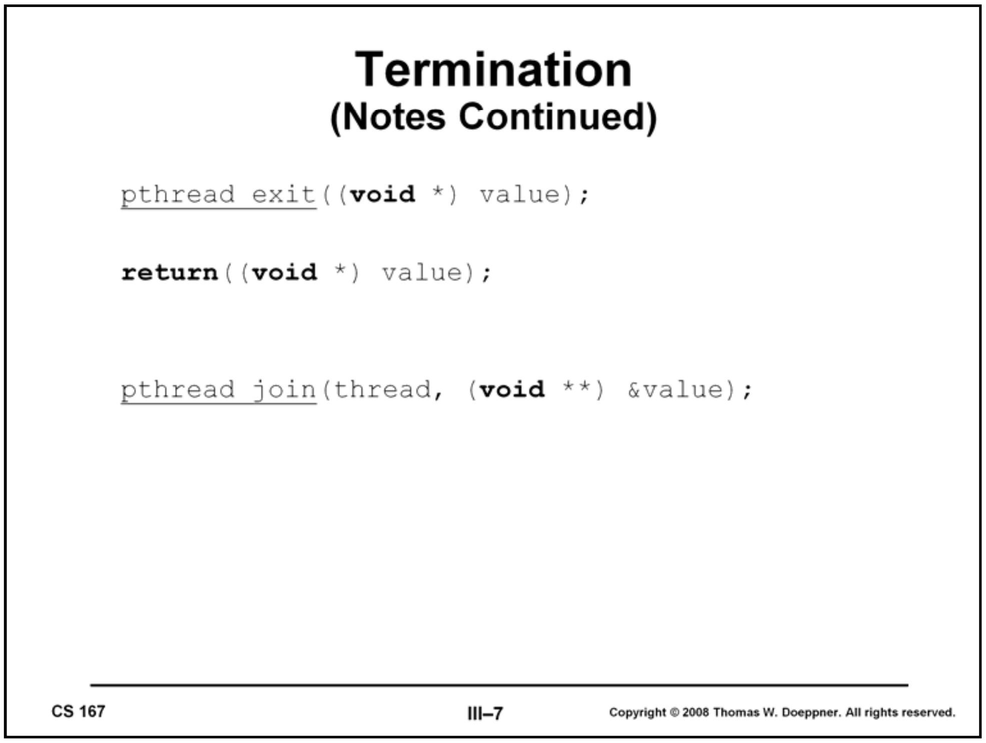
Termination

```
pthread_exit((void *) value);

return((void *) value);


pthread_join(thread, (void **) &value);
```

CS 167 III–6 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

A thread terminates either by calling *pthread_exit* or by returning from its first procedure. In either case, it supplies a value that can be retrieved via a call (by some other thread) to *pthread_join*. The analogy to process termination and the waitpid system call in Unix is tempting and is correct to a certain extent—Unix's *wait*, like *pthread_join*, lets one caller synchronize with the termination of another. There is one important difference, however: Unix has the notion of parent/child relationships among processes. A process may wait only for its children to terminate. No such notion of parent/child relationship is maintained with POSIX threads: one thread may wait for the termination of any other thread in the process (though some threads cannot be "joined" by any thread—see the next page). It is, however, important that *pthread_join* be called for each joinable terminated thread—since threads that have terminated but have not yet been joined continue to use up some resources, resources that will be freed once the thread has been joined. The effect of multiple threads calling *pthread_join* is "undefined"—meaning that what happens can vary from one implementation to the next.

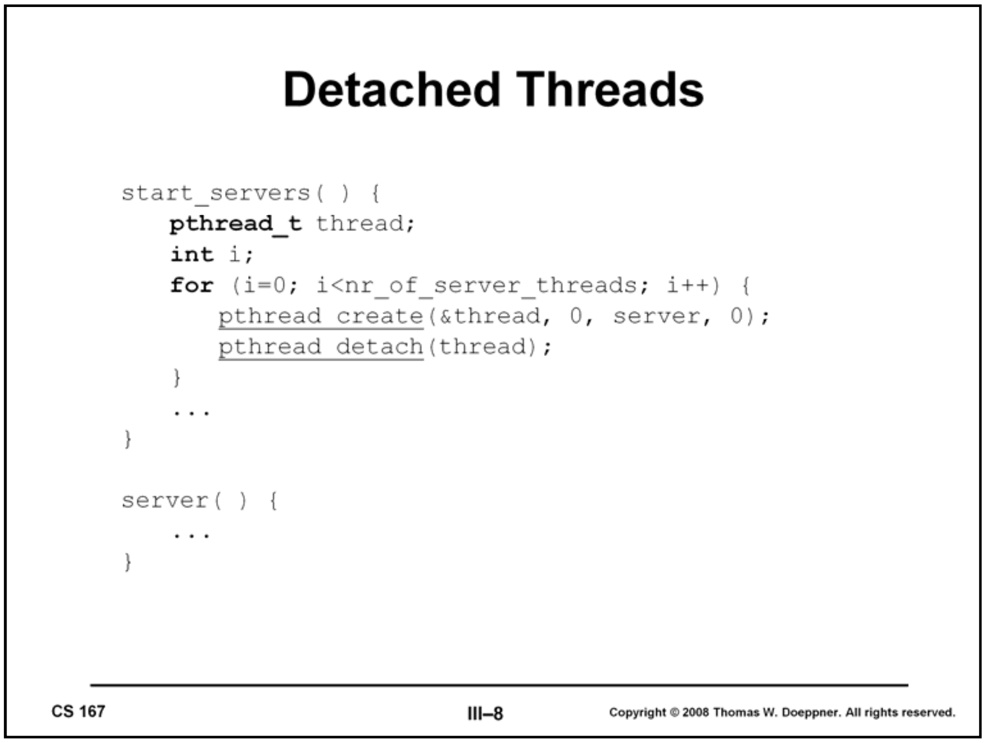
One should be careful to distinguish between terminating a thread and terminating a process. With the latter, all the threads in the process are forcibly terminated. So, if any thread in a process calls exit, the entire process is terminated, along with its threads. Similarly, if a thread returns from main, this also terminates the entire process, since returning from main is equivalent to calling exit. The only thread that can legally return from main is the one that called it in the first place. All other threads (those that did not call main) certainly do not terminate the entire process when they return from their first procedures, they merely terminate themselves.



Termination
(Notes Continued)

```
pthread_exit((void *) value);

return((void *) value);


pthread_join(thread, (void **) &value);
```

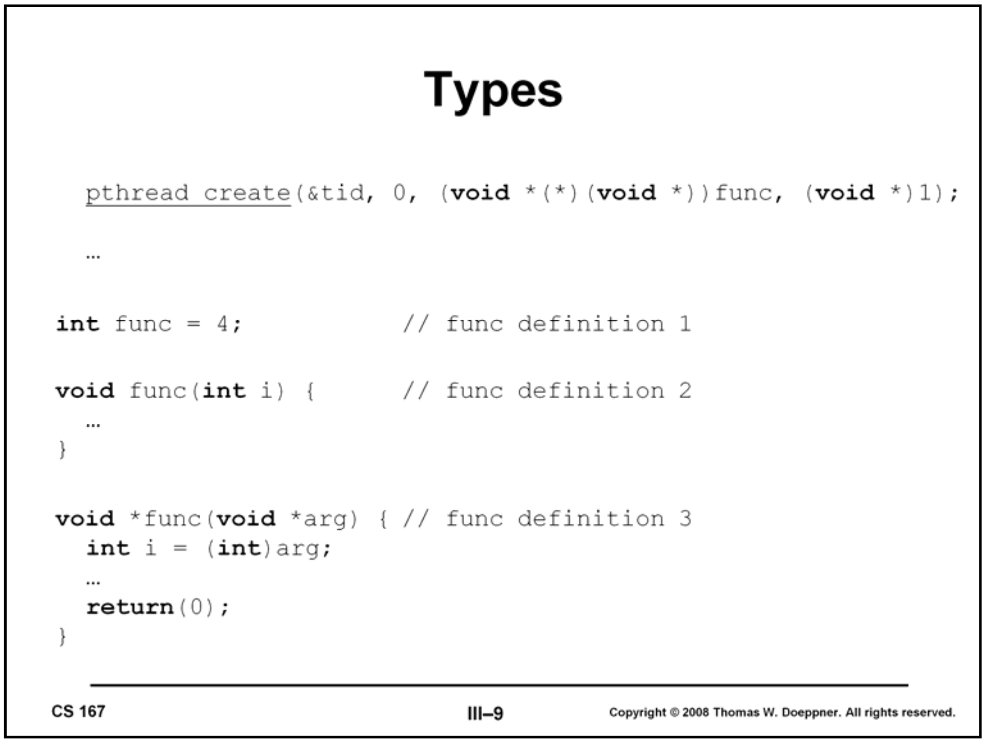
If no thread calls *exit* and no thread returns from main, then the process should terminate once all threads have terminated (i.e., have called *pthread_exit* or, for threads other than the first one, have returned from their first procedure). If the first thread calls *pthread_exit*, it self-destructs, but does not cause the process to terminate (unless no other threads are extant).



Detached Threads

```
start_servers( ) {
    pthread_t thread;
    int i;
    for (i=0; i<nr_of_server_threads; i++) {
        pthread_create(&thread, 0, server, 0);
        pthread_detach(thread);
    }
    ...
}

server( ) {
    ...
}
```

If there is no reason to synchronize with the termination of a thread, then it is rather a nuisance to have to call *pthread_join*. Instead, one can arrange for a thread to be detached. Such threads "vanish" when they terminate—not only do they not need to be joined with, but they cannot be joined with.



Types

```
  pthread_create(&tid, 0, (void *(*)(void *))func, (void *)1);

  …

int func = 4;             // func definition 1

void func(int i) {        // func definition 2
  …
}

void *func(void *arg) { // func definition 3
  int i = (int)arg;
  …
  return(0);
}
```

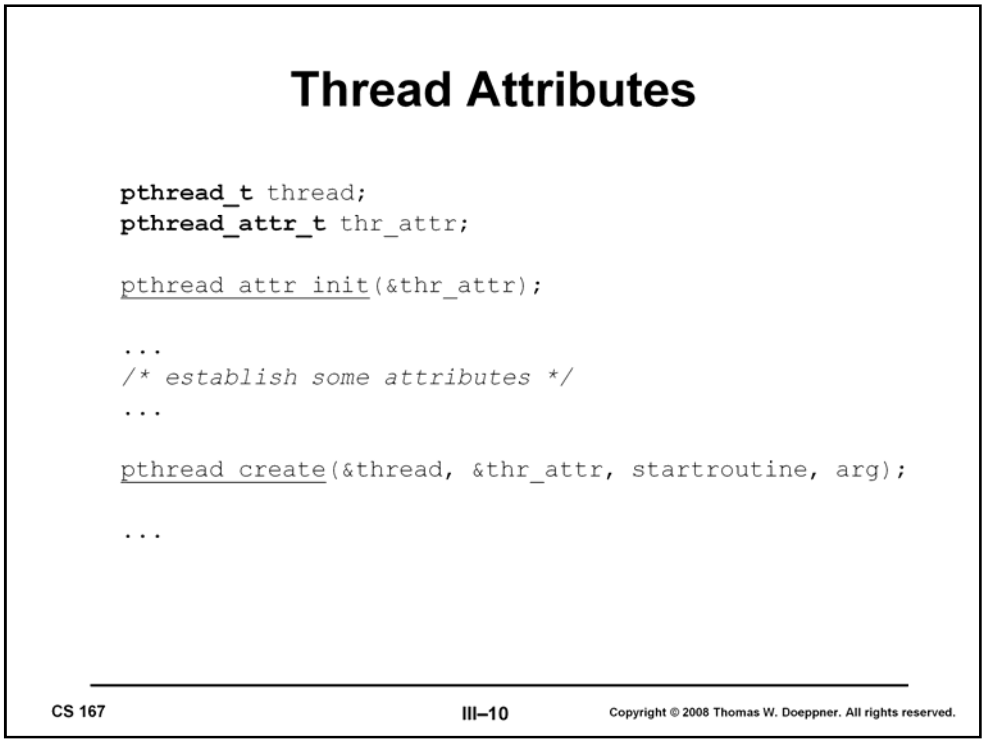
CS 167 III–9 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

This slide shows some problems that can occur because of confusion about types. Suppose *pthread_create* is called, as shown in the slide. The third argument, *func*, has been carefully cast so that the compiler will not issue warning messages. However, if the definition of func is the first one, there will be a serious problem when the program is run! The issue here is that the C compiler trusts you when you give it a cast—it's up to you to make certain that the trust is warranted.

Suppose the second definition of *func* is used. At least this one is a function. However, there are two potential problems. The given definition returns nothing, though *pthread_create* assumes it returns a *void **. On most, if not all of today's systems, this probably is not a problem, though it is conceivable that the calling sequence for a function that returns an argument is different from that for a function that doesn't (e.g., space for the return value might be allocated on the stack).

The second problem can definitely occur on some of today's architectures. The argument being passed to *func* is supplied as an *int*, though it will be passed as a *void **. The routine *func*, however, expects an *int*, though receives a *void **. If there is no difference in size between an *int* and a *void **, this is not likely to be a problem. However, consider a 64-bit machine on which a *void ** occupies 64 bits and an *int* occupies 32 bits. In our example, the 32-bit *int* would be passed within a 64-bit *void **, probably as the least-significant bits; *func* would receive a 64-bit quantity, but would use only 32 bits of it. Which 32 bits? On little-endian architectures, *func* would use the least significant 32 bits and things would work, but there would be problems on big-endian architectures, where the most significant bits would be used.

The correct way of doing things is shown in the third definition, in which *func* does return something and there is an explicit (and legal) conversion from *void ** to *int*.



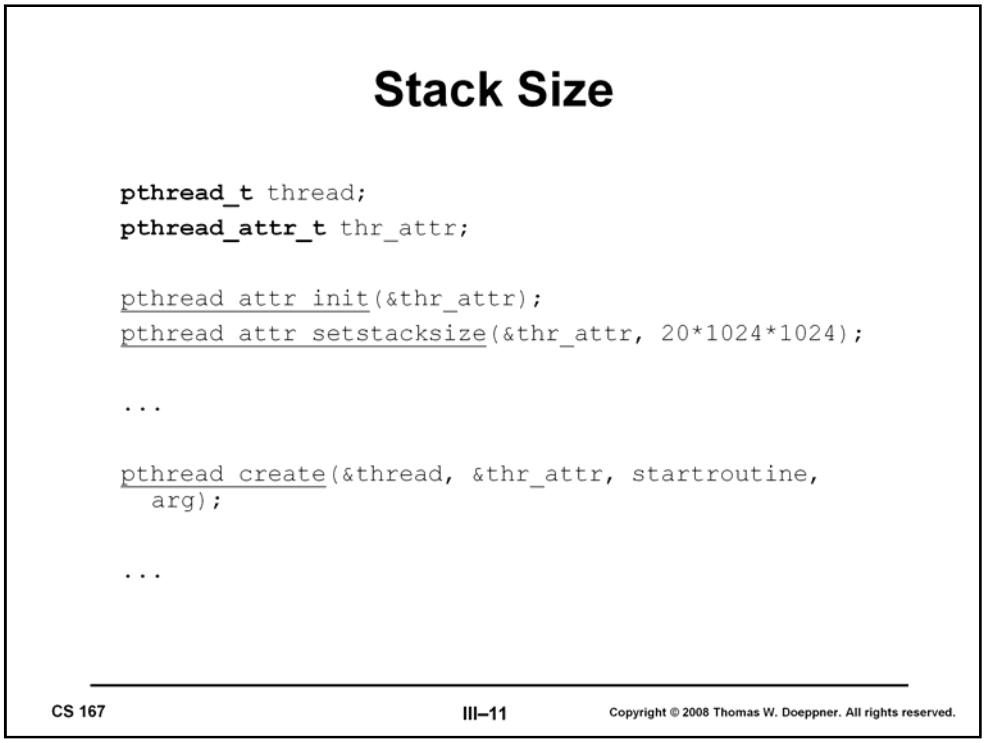
Thread Attributes

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);

...
/* establish some attributes */
...

pthread_create(&thread, &thr_attr, startroutine, arg);

...
```

A number of properties of a thread can be specified via the attributes argument when the thread is created. Some of these properties are specified as part of the POSIX specification, others are left up to the implementation. By burying them inside the attributes structure, we make it straightforward to add new types of properties to threads without having to complicate the parameter list of *pthread_create*. To set up an attributes structure, one must call *pthread_attr_init*. As seen in the next slide, one then specifies certain properties, or attributes, of threads. One can then use the attributes structure as an argument to the creation of any number of threads.

Note that the attributes structure only affects the thread when it is created. Modifying an attributes structure has no effect on already-created threads, but only on threads created subsequently with this structure as the attributes argument.

Storage may be allocated as a side effect of calling *pthread_attr_init*. To ensure that it is freed, call *pthread_attr_destroy* with the attributes structure as argument. Note that if the attributes structure goes out of scope, not all storage associated with it is necessarily released—to release this storage you must call *pthread_attr_destroy*.

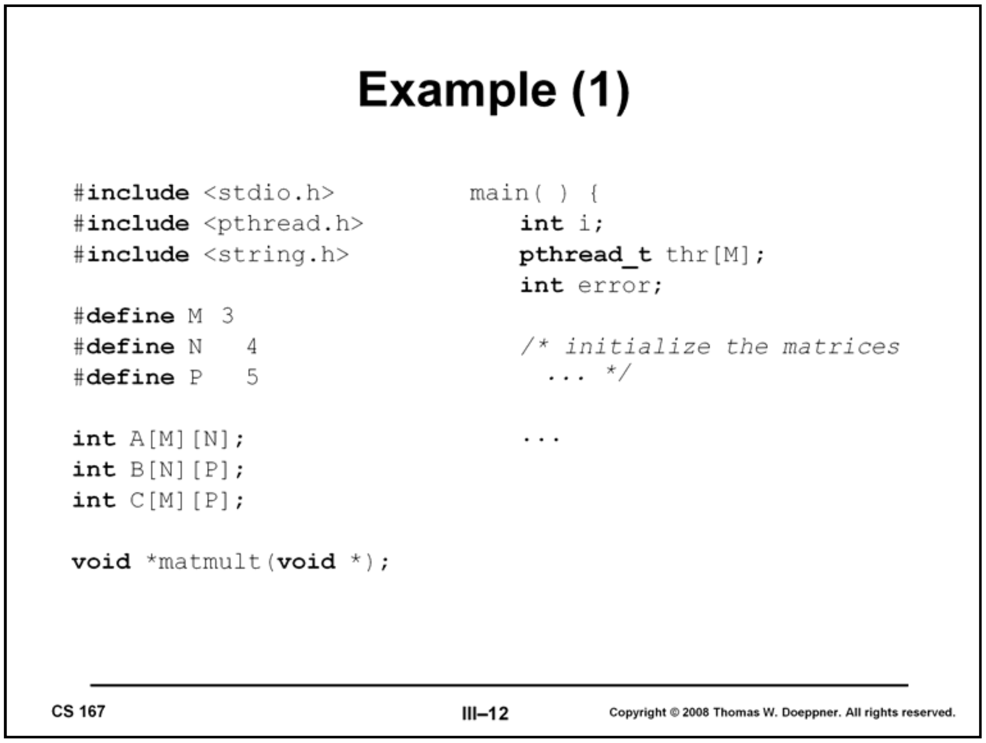


Stack Size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);

...

pthread_create(&thread, &thr_attr, startroutine,
  arg);

...
```

Among the attributes that can be specified is a thread's stack size. The default attributes structure specifies a stack size that is probably good enough for "most" applications. How big is it? The default stack size is not mandated by POSIX. In Digital Unix 4.0, the default stack size is 21,120 bytes, while in Solaris it is one megabyte. To establish a different stack size, use the *pthread_attr_setstacksize* routine, as shown in the slide.

How large a stack is necessary? The answer, of course, is that it depends. If the stack size is too small, there is the danger that a thread will attempt to overwrite the end of its stack. There is no problem with specifying too large a stack, except that, on a 32-bit machine, one should be careful about using up too much address space (one thousand threads, each with a megabyte stack, use a fair portion of the address space).



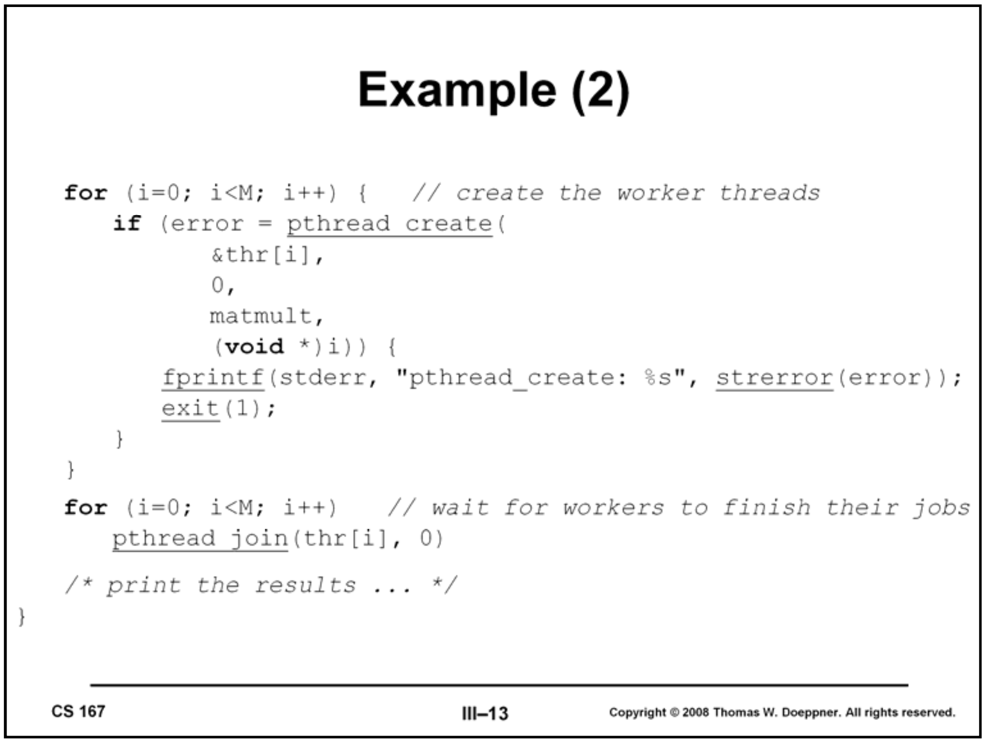
Example (1)

```
#include <stdio.h>          main( ) {
#include <pthread.h>            int i;
#include <string.h>             pthread_t thr[M];
                                int error;
#define M 3
#define N   4                   /* initialize the matrices
#define P   5                     ... */

int A[M][N];                    ...
int B[N][P];
int C[M][P];

void *matmult(void *);
```

CS 167 III–12 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

In this series of slides we present a simple example of an (almost) complete program—one that multiplies two matrices using a number of threads. Our algorithm is not an example of an efficient matrix-multiplication algorithm, but it shows us everything that must be included to make a multithreaded C program compile and run. Our approach is to use the most straightforward technique for multiplying two matrices: each element of the product is formed by directly taking the inner product of a row of the multiplier and a column of the multiplicand. We employ multiple threads by assigning a thread to compute each row of the product.

This slide shows the necessary includes, global declarations, and the beginning of the main routine.



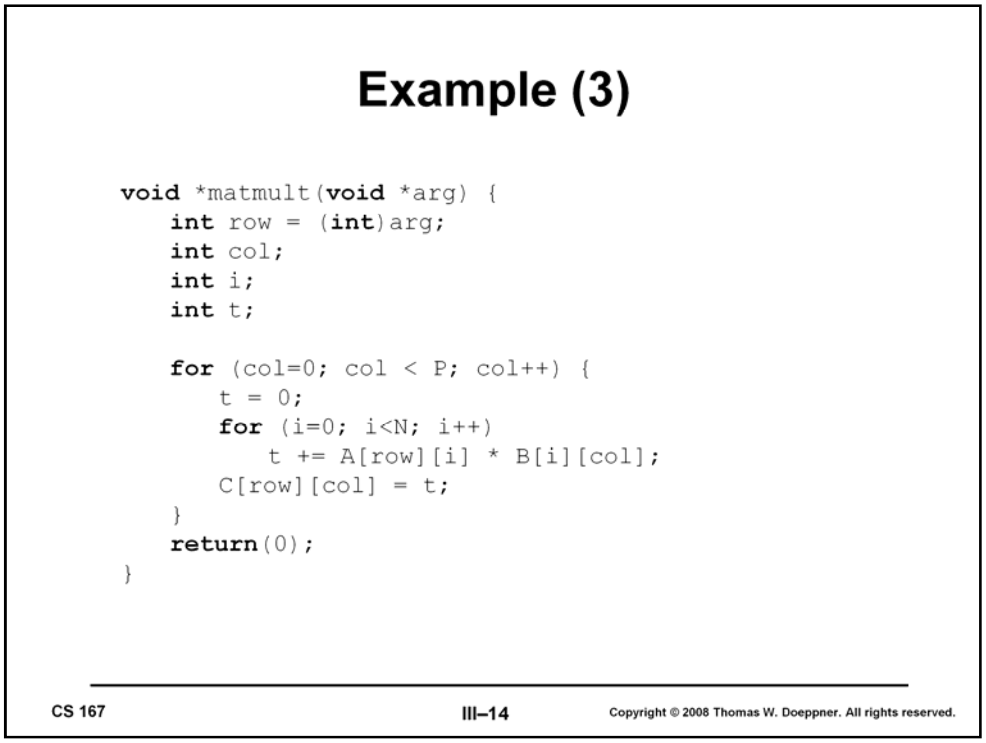
Example (2)

```
   for (i=0; i<M; i++) {   // create the worker threads
      if (error = pthread_create(
             &thr[i],
             0,
             matmult,
             (void *)i)) {
         fprintf(stderr, "pthread_create: %s", strerror(error));
         exit(1);
      }
   }
   for (i=0; i<M; i++)   // wait for workers to finish their jobs
      pthread_join(thr[i], 0)
   /* print the results ... */
}
```

CS 167 III–13 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

Here we have the remainder of main. It creates a number of threads, one for each row of the result matrix, waits for all of them to terminate, then prints the results (this last step is not spelled out). Note that we check for errors when calling *pthread_create*. (It is important to check for errors after calls to almost all of the pthread routines, but we normally omit it in the slides for lack of space.) For reasons discussed later, the pthread calls, unlike Unix system calls, do not return -1 if there is an error, but return the error code itself (and return zero on success). However, one can find the text associated with error codes, just as for Unix-system-call error codes, by calling *strerror*.

So that the first thread is certain that all the other threads have terminated, it must call *pthread_join* on each of them.

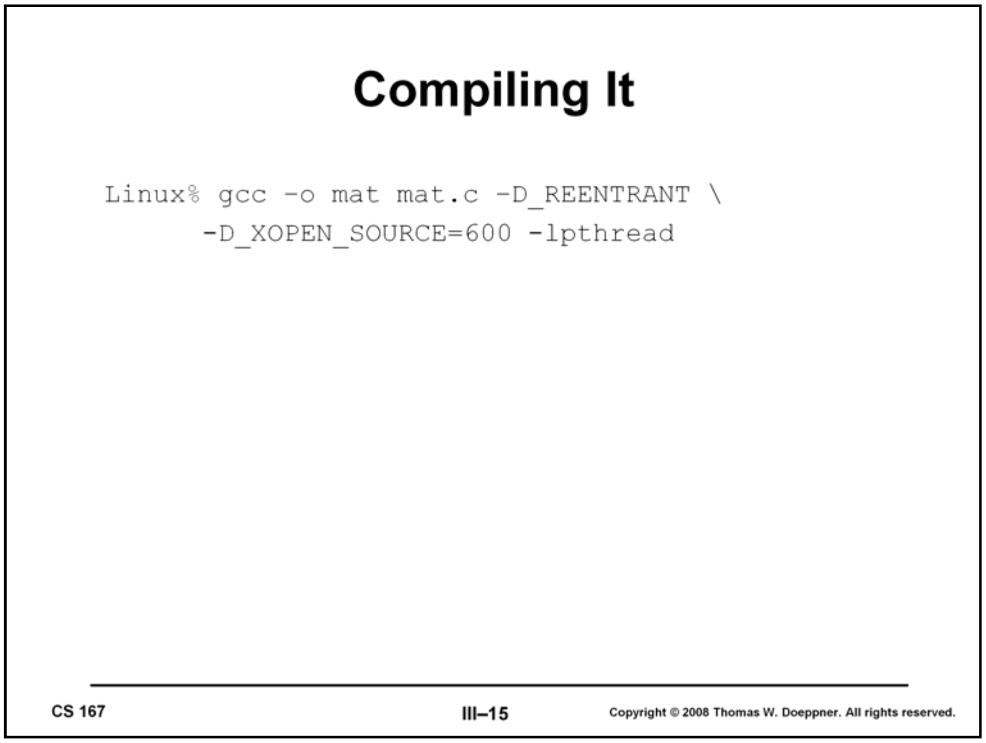


Example (3)

```
void *matmult(void *arg) {
    int row = (int)arg;
    int col;
    int i;
    int t;

    for (col=0; col < P; col++) {
        t = 0;
        for (i=0; i<N; i++)
            t += A[row][i] * B[i][col];
        C[row][col] = t;
    }
    return(0);
}
```

Here is the code executed by each of the threads. It's pretty straightforward: it merely computes a row of the result matrix.

Note how the argument is explicitly converted from *void* * to *int*.



Compiling It

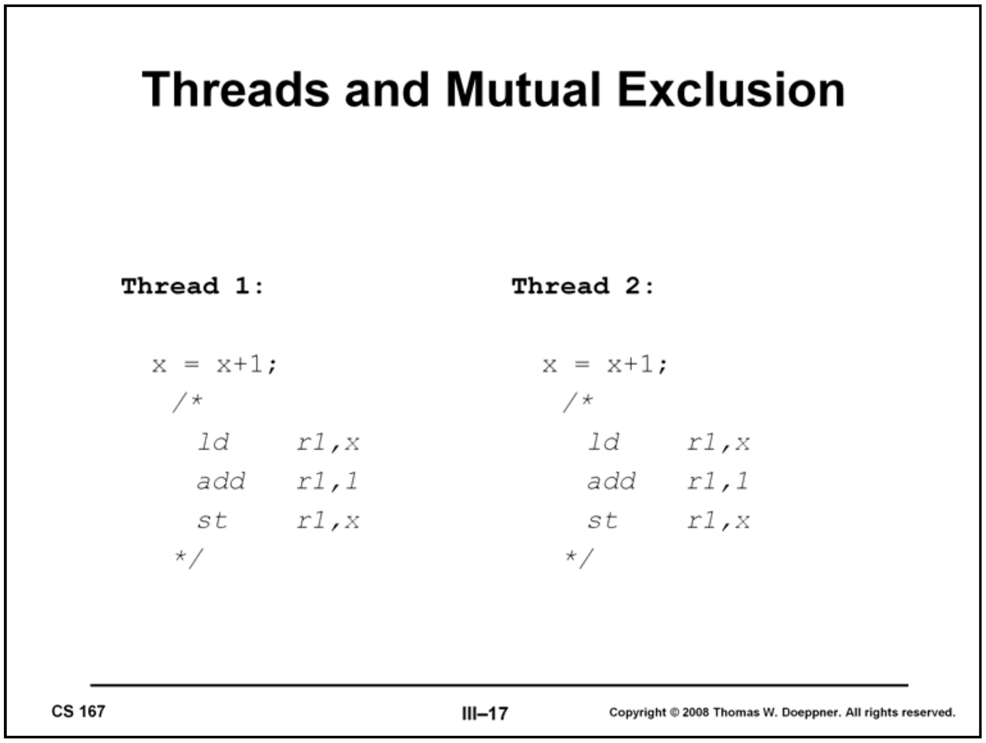
```
Linux% gcc -o mat mat.c -D_REENTRANT \
      -D_XOPEN_SOURCE=600 -lpthread
```

Here is how to compile a multithreaded C program in Linux. The definition of _XOPEN_SOURCE is required to get “XOPEN single Unix” conformance, which is required for a number of POSIX-threads features.

Mutual Exclusion

CS 167 III–16 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

The mutual-exclusion problem involves making certain that two things don't happen at once. A non-computer example arose in the fighter aircraft of World War I. Due to a number of constraints (e.g., machine guns tended to jam frequently and thus had to be close to people who could unjam them), machine guns were mounted directly in front of the pilot. However, blindly shooting a machine gun through the whirling propeller was not a good idea—one was apt to shoot oneself down. At the beginning of the war, pilots politely refrained from attacking fellow pilots. A bit later in the war, however, the Germans developed the tactic of gaining altitude on an opponent, diving at him, turning off the engine, then firing without hitting the now-stationary propeller. Today, this would be called *coarse-grained synchronization*. Later, the Germans developed technology that synchronized the firing of the gun with the whirling of the propeller, so that shots were fired only when the propeller blades would not be in the way. This is perhaps the first example of a mutual-exclusion mechanism providing *fine-grained synchronization*.



Threads and Mutual Exclusion

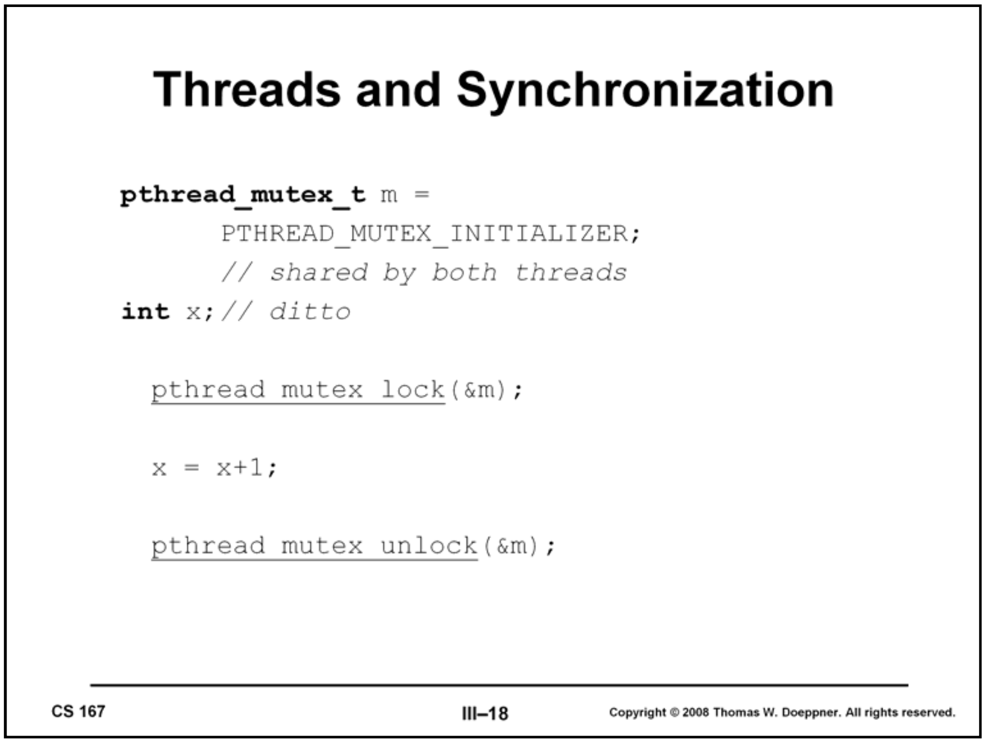
Thread 1:

```
x = x+1;
 /*
   ld    r1,x
   add   r1,1
   st    r1,x
 */
```

Thread 2:

```
x = x+1;
 /*
   ld    r1,x
   add   r1,1
   st    r1,x
 */
```

Here we have two threads that are reading and modifying the same variable: both are adding one to *x*. Although the operation is written as a single step in terms of C code, it generally takes three machine instructions, as shown in the slide. If the initial value of *x* is 0 and the two threads execute the code shown in the slide, we might expect that the final value of *x* is 2. However, suppose the two threads execute the machine code at roughly the same time: each loads the value of *x* into its register, each adds one to the contents of the register, and each stores the result into *x*. The final result, of course, is that *x* is 1, not 2.



Threads and Synchronization

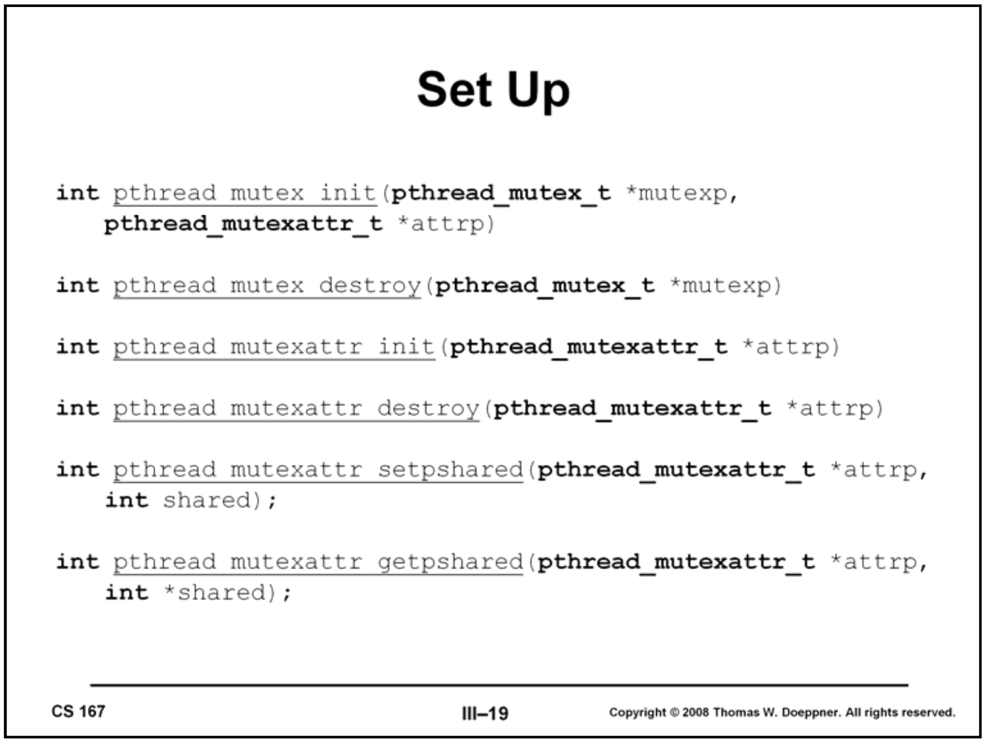
```
pthread_mutex_t m =
      PTHREAD_MUTEX_INITIALIZER;
      // shared by both threads
int x;// ditto

  pthread_mutex_lock(&m);

  x = x+1;

  pthread_mutex_unlock(&m);
```

CS 167 III–18 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

To solve our synchronization problem, we introduce mutexes—a synchronization construct providing mutual exclusion. A mutex is used to insure either that only one thread is executing a particular piece of code at once (code locking) or that only one thread is accessing a particular data structure at once (data locking). A mutex belongs either to a particular thread or to no thread (i.e., it is either locked or unlocked). A thread may lock a mutex by calling *pthread_mutex_lock*. If no other thread has the mutex locked, then the calling thread obtains the lock on the mutex and returns. Otherwise it waits until no other thread has the mutex, and finally returns with the mutex locked. There may of course be multiple threads waiting for the mutex to be unlocked. Only one thread can lock the mutex at a time; there is no specified order for who gets the mutex next, though the ordering is assumed to be at least somewhat “fair.”

To unlock a mutex, a thread calls *pthread_mutex_unlock*. It is considered incorrect to unlock a mutex that is not held by the caller (i.e., to unlock someone else’s mutex). However, it is somewhat costly to check for this, so most implementations, if they check at all, do so only when certain degrees of debugging are turned on.

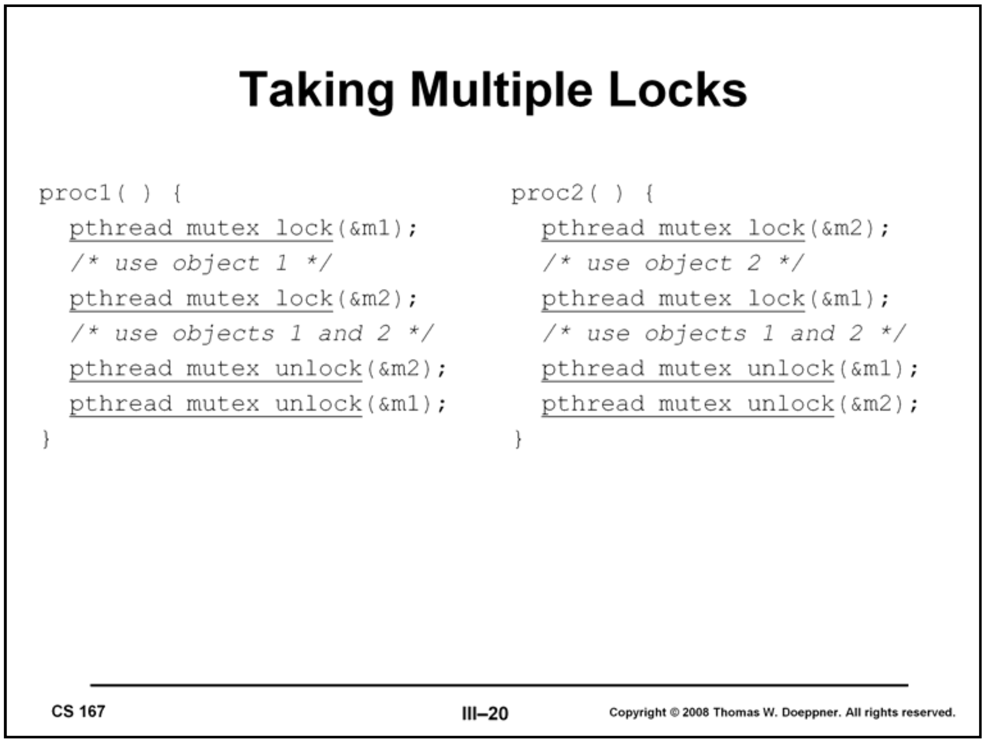
Like any other data structure, mutexes must be initialized. This can be done via a call to *pthread_mutex_init* or can be done statically by assigning PTHREAD_MUTEX_INITIALIZER to a mutex. The initial state of such initialized mutexes is unlocked. Of course, a mutex should be initialized only once! (I.e., make certain that, for each mutex, no more than one thread calls *pthread_mutex_init*.)



Set Up

```
int pthread_mutex_init(pthread_mutex_t *mutexp,
   pthread_mutexattr_t *attrp)

int pthread_mutex_destroy(pthread_mutex_t *mutexp)

int pthread_mutexattr_init(pthread_mutexattr_t *attrp)

int pthread_mutexattr_destroy(pthread_mutexattr_t *attrp)

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attrp,
   int shared);

int pthread_mutexattr_getpshared(pthread_mutexattr_t *attrp,
   int *shared);
```

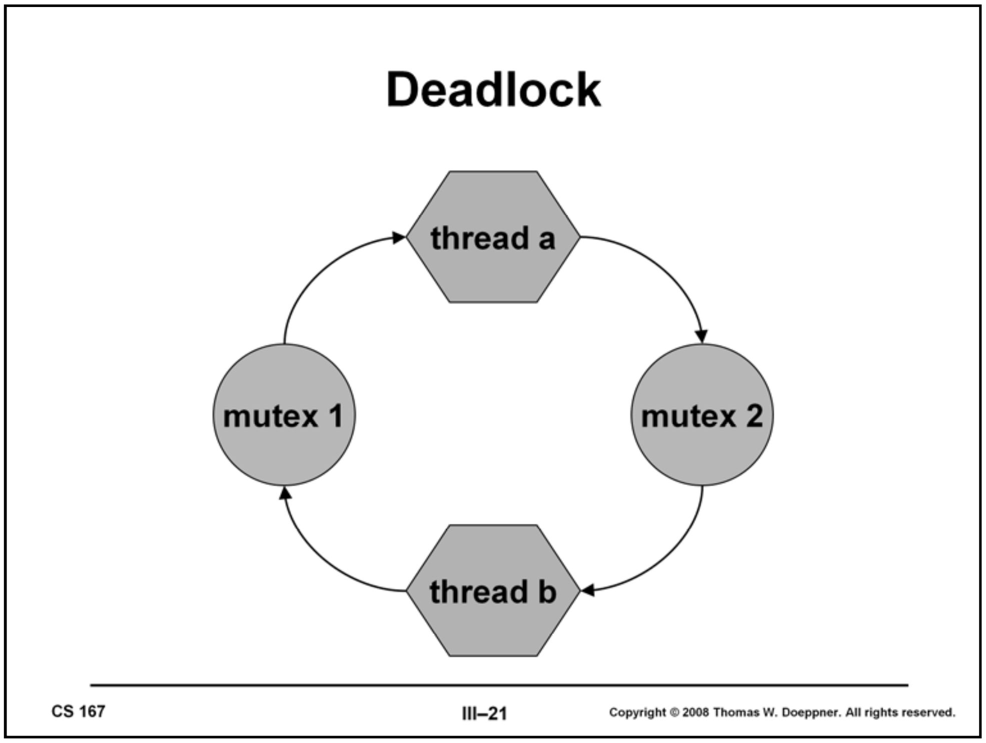
The routines *pthread_mutex_init* and *pthread_mutex_destroy* are supplied to initialize and to destroy a mutex. (They do not allocate or free the storage for the primary mutex data structure, but they might allocate and free storage referred to by the mutex data structure.) As with threads, an attribute structure encapsulates the various parameters that might apply to the mutex. The routines *pthread_mutexattr_init* and *pthread_mutexattr_destroy* control the initialization and destruction of these attribute structures. One standard attribute is whether a mutex is for use only by threads in one process or for use by threads in multiple processes (in which case the mutex must be allocated in a region of shared memory). The former is the default, which can also be specified by passing an attributes structure to *pthread_mutex_init* that has first been initialized via a call to *pthread_mutexattr_init* and then modified via a call to *pthread_mutexattr_setpshared* with *shared* set to PTHREAD_PROCESS_PRIVATE. To get the latter behavior, one instead calls *pthread_mutexattr_setpshared* with *shared* set to PTHREAD_PROCESS_SHARED. However, this behavior, though supported on Solaris, is not supported on Linux, at least through Linux 2.4.



Taking Multiple Locks

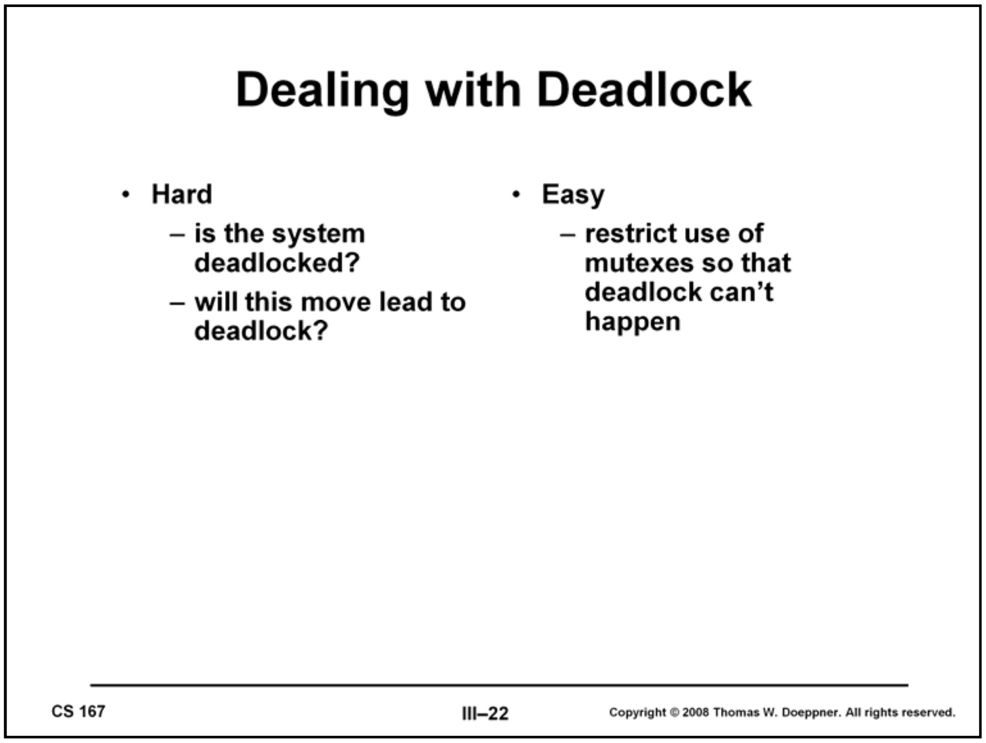
```
proc1( ) {                             proc2( ) {
  pthread_mutex_lock(&m1);               pthread_mutex_lock(&m2);
  /* use object 1 */                     /* use object 2 */
  pthread_mutex_lock(&m2);               pthread_mutex_lock(&m1);
  /* use objects 1 and 2 */              /* use objects 1 and 2 */
  pthread_mutex_unlock(&m2);             pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m1);             pthread_mutex_unlock(&m2);
}                                      }
```

In this example our threads are using two mutexes to control access to two different objects. Thread 1, executing *proc1*, first takes mutex 1, then, while still holding mutex 1, obtains mutex 2. Thread 2, executing *proc2*, first takes mutex 2, then, while still holding mutex 2, obtains mutex 1. However, things do not always work out as planned. If thread 1 obtains mutex 1 and, at about the same time, thread 2 obtains mutex 2, then if thread 1 attempts to take mutex 2 and thread 2 attempts to take mutex 1, we have a *deadlock*.



Deadlock

The slide shows what's known as a *resource graph*: a directed graph with two sorts of nodes, representing threads and mutexes (protecting resources). There's an arc from a mutex to a thread if the thread has that mutex locked. There's an arc from a thread to a mutex if the thread is waiting to lock that mutex. Clearly, such a graph has a cycle if and only if there is a deadlock.



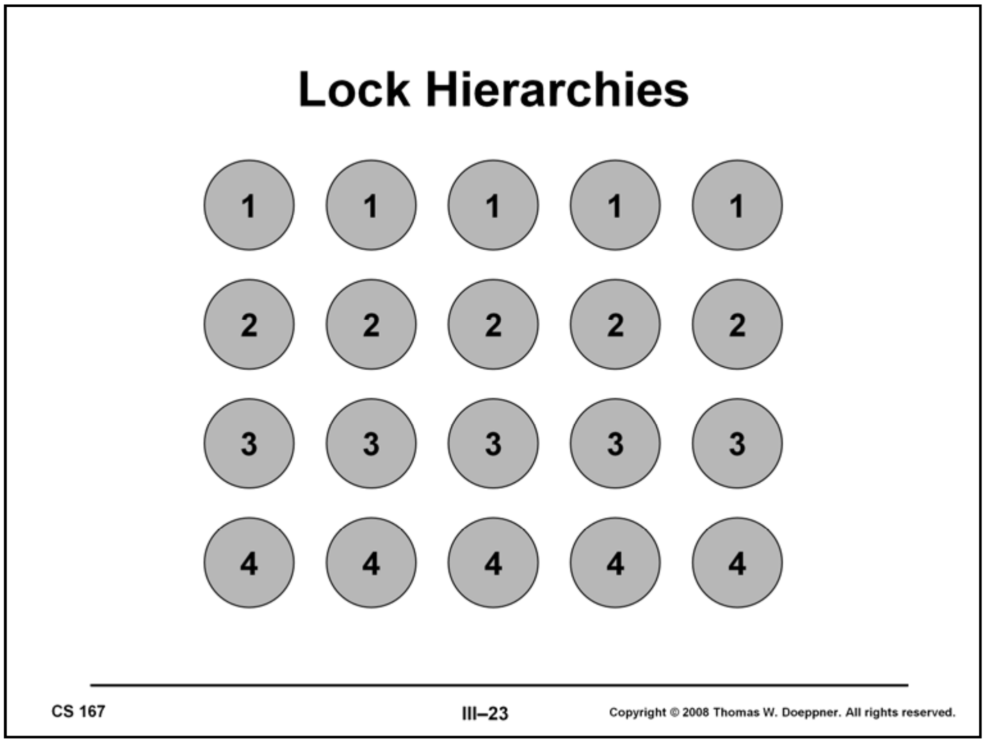
Dealing with Deadlock

- Hard
  - is the system deadlocked?
  - will this move lead to deadlock?
- Easy
  - restrict use of mutexes so that deadlock can't happen

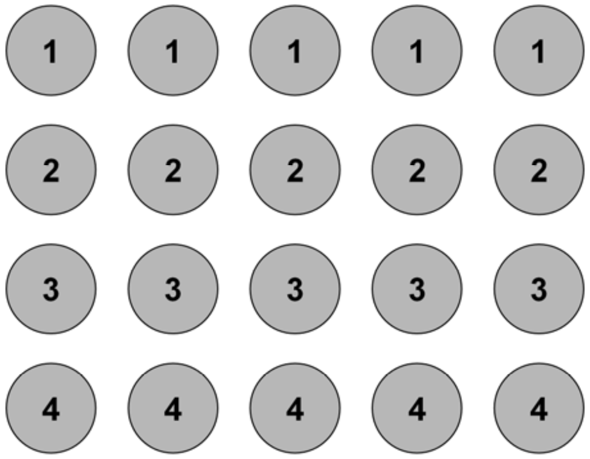
CS 167 III–22 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

The general problems of detecting deadlocked situations and avoiding deadlock are difficult. No algorithms exist that are of practical use for multithreaded code (note that even an algorithm whose running time is linear in the total number of nodes would be too expensive). See the textbook for a discussion of some non-practical algorithms.

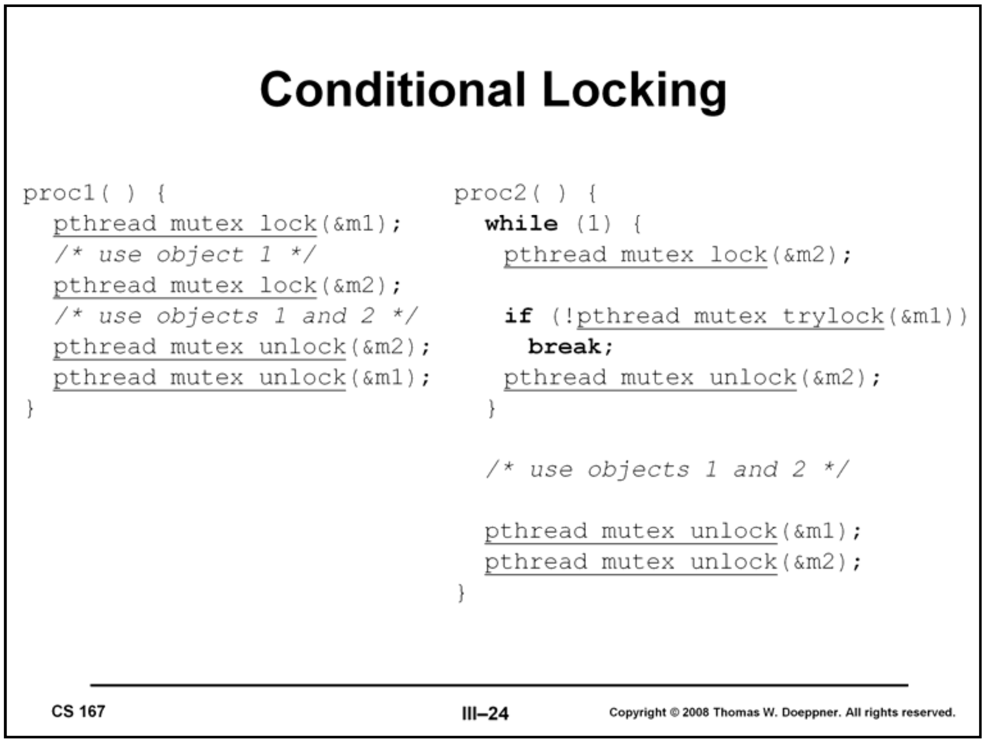
However, by restricting the use of mutexes such that threads locking multiple mutexes must do so in a prescribed order, we can assure that there are no cycles in the resource graph and thus no chance of deadlock. Fortunately, this restriction can normally be easily made.



Lock Hierarchies



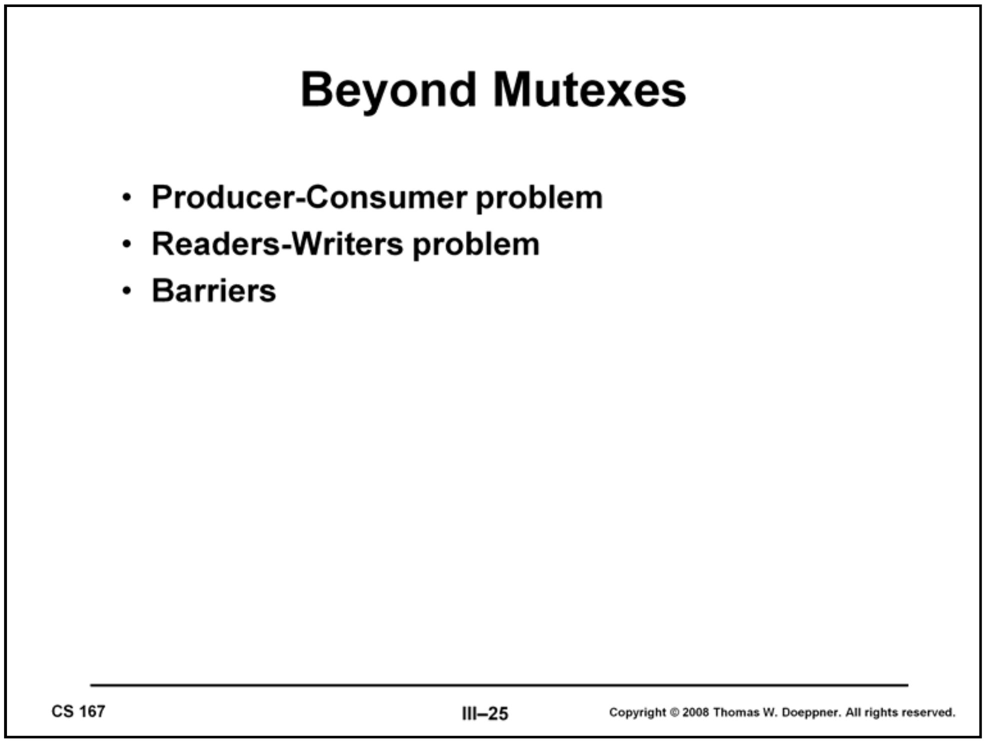
One easy approach for assuring that mutexes are taken in a prescribed order is to organize them into a hierarchy. If a thread is holding a lock at level *i*, it must not request another lock at level *i* or less: it must request only locks at levels greater than *i*.



Conditional Locking

```
proc1( ) {                        proc2( ) {
  pthread_mutex_lock(&m1);          while (1) {
  /* use object 1 */                  pthread_mutex_lock(&m2);
  pthread_mutex_lock(&m2);
  /* use objects 1 and 2 */           if (!pthread_mutex_trylock(&m1))
  pthread_mutex_unlock(&m2);            break;
  pthread_mutex_unlock(&m1);          pthread_mutex_unlock(&m2);
}                                   }

                                    /* use objects 1 and 2 */

                                    pthread_mutex_unlock(&m1);
                                    pthread_mutex_unlock(&m2);
                                  }
```

As we've just discussed, the simplest (and best) approach to avoiding deadlock to make certain that all threads that will hold multiple mutexes simultaneously obtain these mutexes in a prescribed order. Ordinarily this can be done, but occasionally it might turn out to be impossible. For example, we might not know which mutex to take second until the first mutex has already been obtained. To avoid deadlock in such situations, we can use the approach shown in the slide. Here thread 1, executing *proc1*, obtains the mutexes in the correct order. Thread 2, executing *proc2*, must for some reason take the mutexes out of order. If it is holding mutex 2, it must be careful about taking mutex 1. So, rather than call *pthread_mutex_lock*, it calls *pthread_mutex_trylock*, which always returns without blocking. If the mutex is available, *pthread_mutex_trylock* locks the mutex and returns 0. If the mutex is not available (i.e., it is locked by another thread), then *pthread_mutex_trylock* returns a nonzero error code (EBUSY). In the example, if mutex 1 is not available, it is probably because it is currently held by thread 1. If thread 2 were to block waiting for the mutex, we have an excellent chance for deadlock. So, rather than block, thread 1 not only quits trying for mutex 1 but also unlocks mutex 2 (since thread 1 could well be waiting for it). It then starts all over again, first taking mutex 2, then mutex 1.

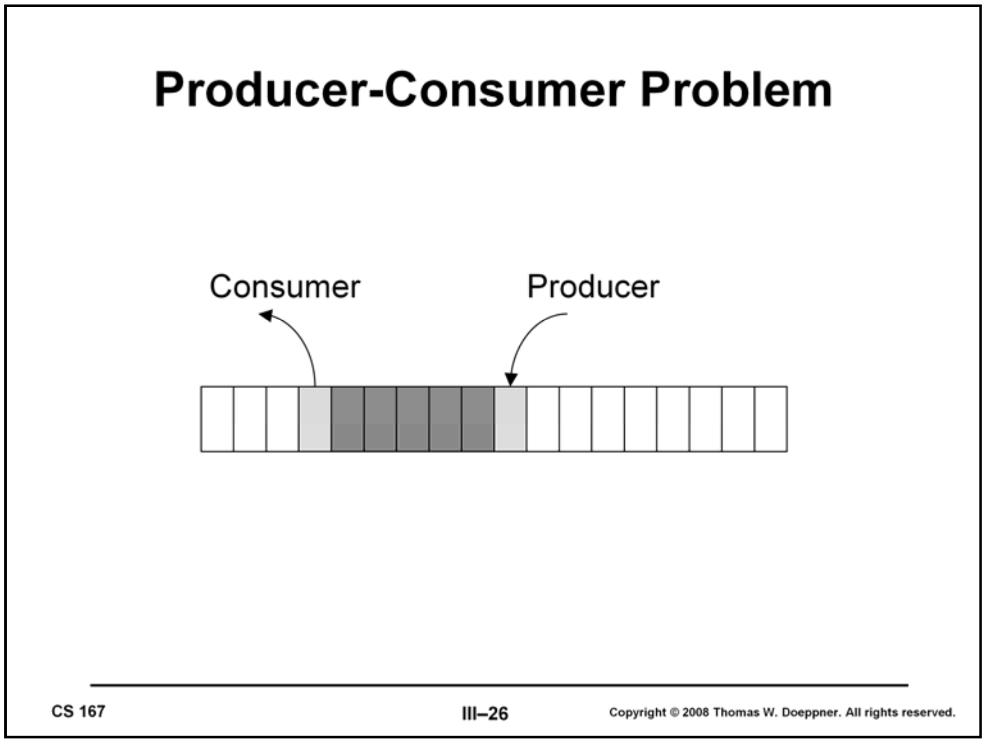


Beyond Mutexes

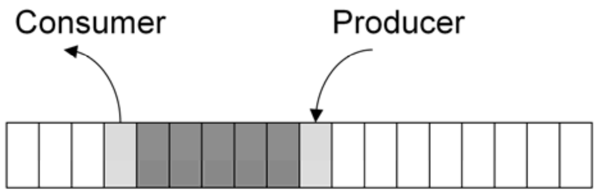
- Producer-Consumer problem
- Readers-Writers problem
- Barriers

CS 167 III–25 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

There are many synchronization problems that cannot be solved with mutexes alone. For example, some require that code be executed only when certain conditions are true. With a mutex, we can safely test if something is true, but we can't arrange for a thread to go to sleep if the condition is false and then be woken up when the condition becomes true. We'll soon see a number of examples where such conditional waiting is needed.

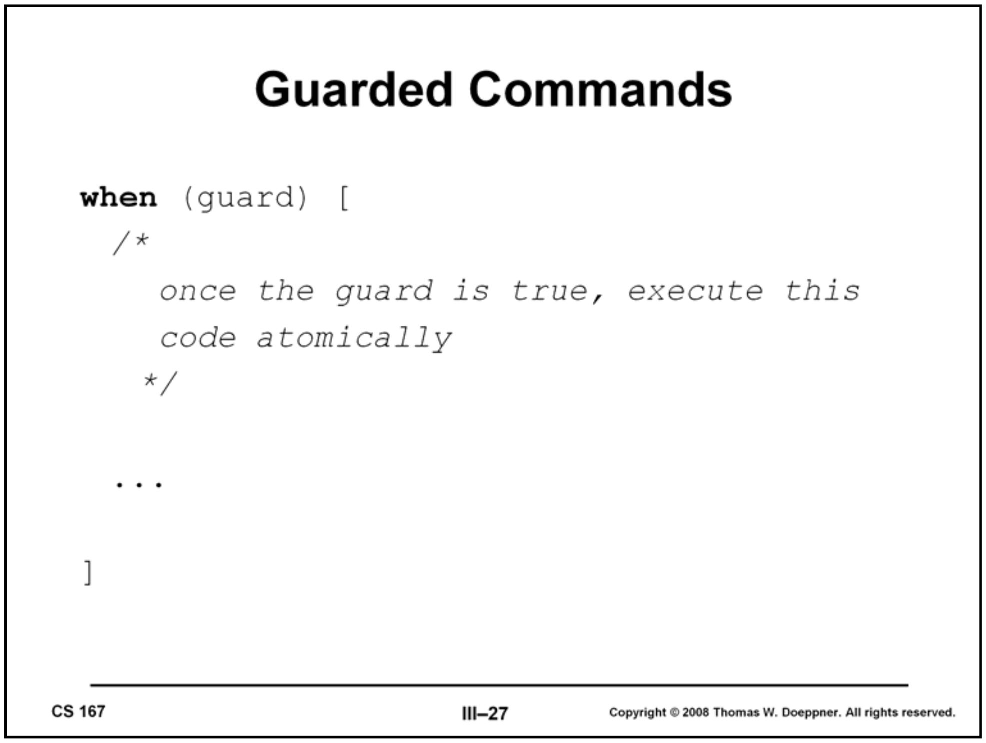


Producer-Consumer Problem



CS 167 III–26 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

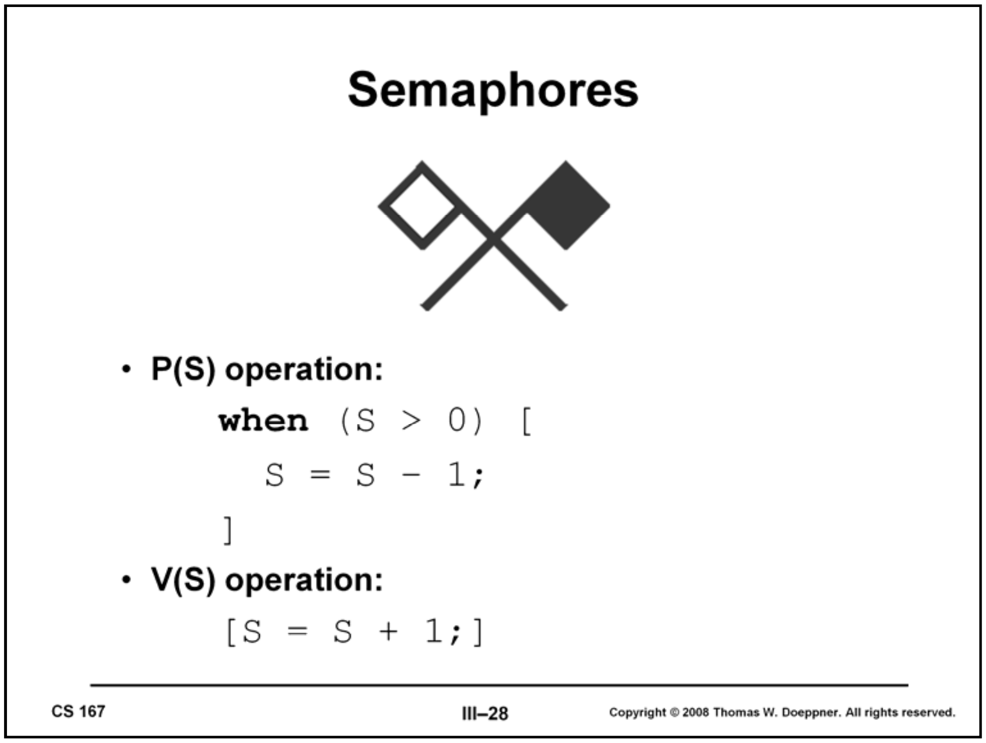
In the *producer-consumer problem* we have two classes of threads, producers and consumers, and a buffer containing a fixed number of slots. A producer thread attempts to put something into the next empty buffer slot, a consumer thread attempts to take something out of the next occupied buffer slot. The synchronization conditions are that producers cannot proceed unless there are empty slots and consumers cannot proceed unless there are occupied slots.



Guarded Commands

```
when (guard) [
  /*
    once the guard is true, execute this
    code atomically
   */

  ...

]
```

Illustrated in the slide is a simple pseudocode construct, the *guarded command*, that we use to describe how various synchronization operations work. The idea is that the code within the square brackets is executed only when the guard (which could be some arbitrary boolean expression) evaluates to true. Furthermore, this code within the square brackets is executed atomically, i.e., the effect is that nothing else happens in the program while the code is executed. Note that the code is not necessarily executed as soon as the guard evaluates to true: we are assured only that when execution of the code begins, the guard is true.



Semaphores

- **P(S) operation:**

```
when (S > 0) [
   S = S - 1;
]
```

- **V(S) operation:**

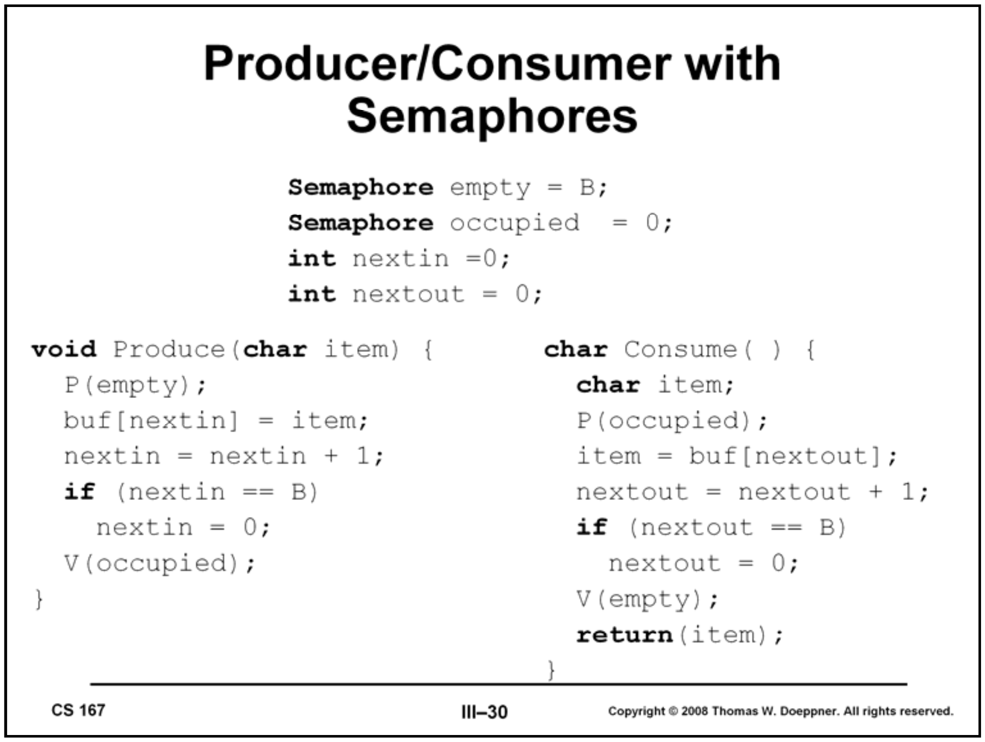
```
[S = S + 1;]
```

Another synchronization construct is the semaphore, designed by Edsger Dijkstra in the 1960s. A semaphore behaves as if it were a nonnegative integer, but it can be operated on only by the semaphore operations. Dijkstra defined two of these: P (for *prolagen*, a made-up word derived from *proberen te verlagen*, which means “try to decrease” in Dutch) and V (for *verhogen*, “increase” in Dutch). Their semantics are shown in the slide.

We think of operations on semaphores as being a special case of guarded execution—a special case that occurs frequently enough to warrant a highly optimized implementation.

Mutexes with Semaphores

```
semaphore S = 1;

void OneAtATime( ) {
  P(S);
  …
  /* code executed mutually exclusively */
  …
  V(S);
}
```



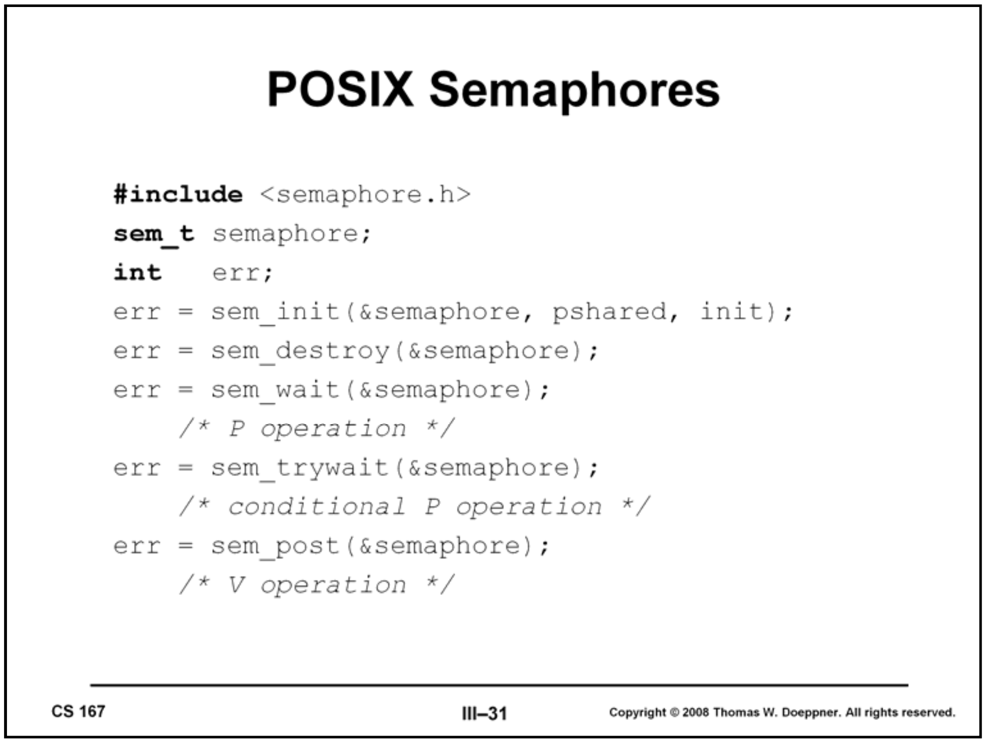
Producer/Consumer with Semaphores

```
Semaphore empty = B;
Semaphore occupied  = 0;
int nextin =0;
int nextout = 0;
```

```
void Produce(char item) {
  P(empty);
  buf[nextin] = item;
  nextin = nextin + 1;
  if (nextin == B)
    nextin = 0;
  V(occupied);
}
```

```
char Consume( ) {
  char item;
  P(occupied);
  item = buf[nextout];
  nextout = nextout + 1;
  if (nextout == B)
    nextout = 0;
  V(empty);
  return(item);
}
```

Here’s a solution for the producer/consumer problem using semaphores—note that it works only with a single producer and a single consumer.

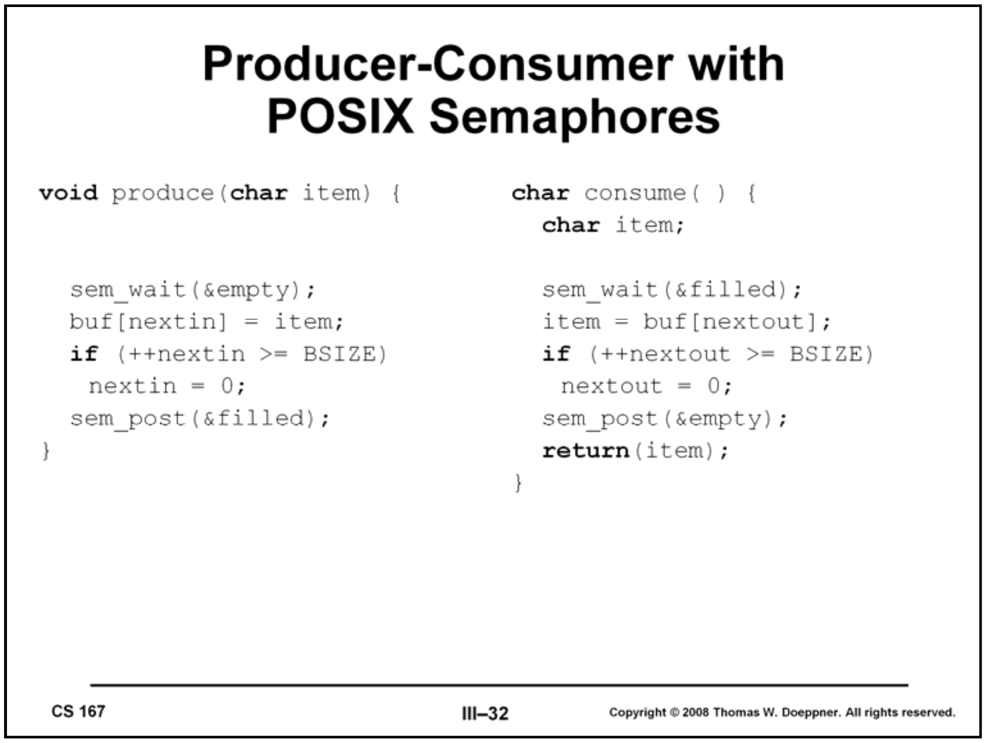


POSIX Semaphores

```
#include <semaphore.h>
sem_t semaphore;
int   err;
err = sem_init(&semaphore, pshared, init);
err = sem_destroy(&semaphore);
err = sem_wait(&semaphore);
    /* P operation */
err = sem_trywait(&semaphore);
    /* conditional P operation */
err = sem_post(&semaphore);
    /* V operation */
```

Here is the POSIX interface for operations on semaphores. (This is not a typo—the blasted "pthread_" prefix really is not used here, since the semaphore operations come from a different POSIX specification—1003.1b. Note also the need for the header file, *semaphore.h*) When creating a semaphore (*sem_init*), rather than supplying an attributes structure, one supplies a single integer argument, *pshared*, which indicates whether the semaphore is to be used only by threads of one process (*pshared = 0*) or by multiple processes (*pshared = 1*). The third argument to *sem_init* is the semaphore's initial value.

All the semaphore operations return zero if successful; otherwise they return an error code.

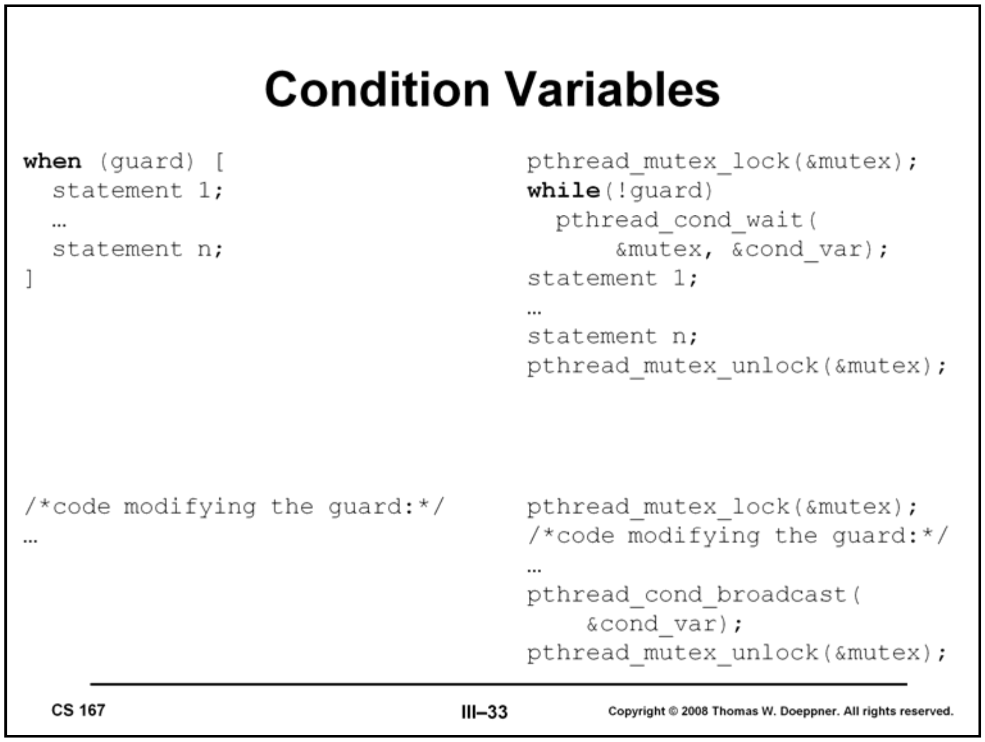


Producer-Consumer with POSIX Semaphores

```
void produce(char item) {


  sem_wait(&empty);
  buf[nextin] = item;
  if (++nextin >= BSIZE)
   nextin = 0;
  sem_post(&filled);
}
```

```
char consume( ) {
  char item;

  sem_wait(&filled);
  item = buf[nextout];
  if (++nextout >= BSIZE)
   nextout = 0;
  sem_post(&empty);
  return(item);
}
```

Here is the producer-consumer solution implemented with POSIX semaphores.



Condition Variables

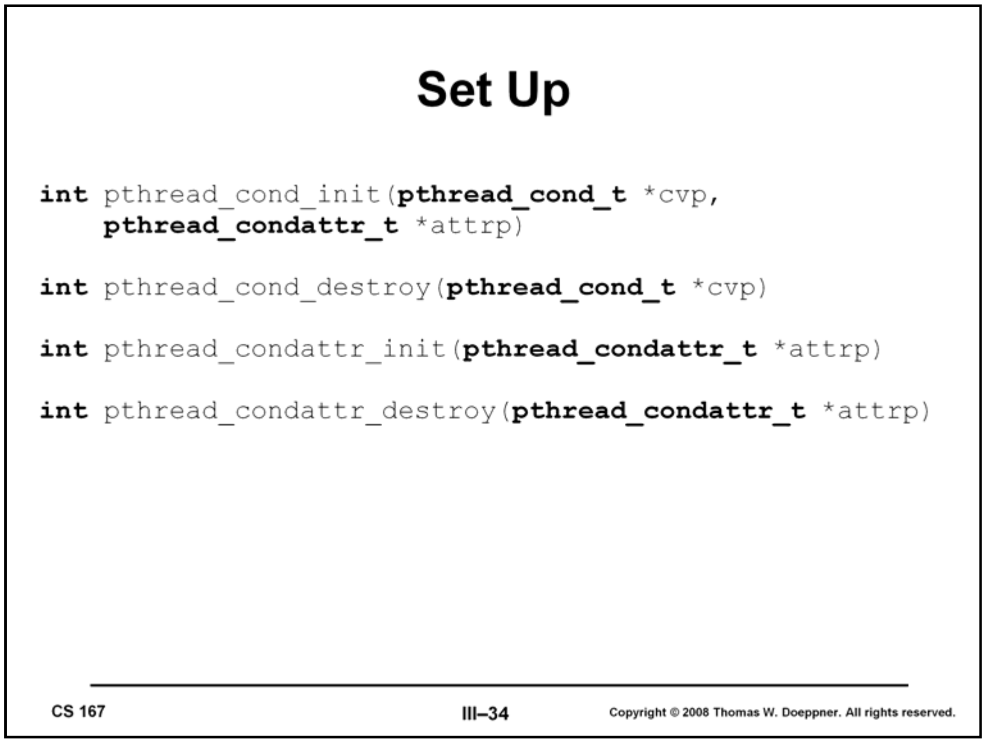
```
when (guard) [                      pthread_mutex_lock(&mutex);
  statement 1;                      while(!guard)
  …                                   pthread_cond_wait(
  statement n;                            &mutex, &cond_var);
]                                   statement 1;
                                    …
                                    statement n;
                                    pthread_mutex_unlock(&mutex);



/*code modifying the guard:*/       pthread_mutex_lock(&mutex);
…                                   /*code modifying the guard:*/
                                    …
                                    pthread_cond_broadcast(
                                        &cond_var);
                                    pthread_mutex_unlock(&mutex);
```

CS 167 III–33 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

Condition variables are another means for synchronization in POSIX; they represent queues of threads waiting to be woken by other threads and can be used to implement guarded commands, as shown in the slide. Though they are rather complicated at first glance, they are even more complicated when you really get into them.

A thread puts itself to sleep and joins the queue of threads associated with a condition variable by calling *pthread_cond_wait*. When it places this call, it must have some mutex locked, and it passes the mutex as the second argument. As part of the call, the mutex is unlocked and the thread is put to sleep, *all in a single atomic step*: i.e., nothing can happen that might affect the thread between the moments when the mutex is unlocked and when the thread goes to sleep. Threads queued on a condition variable are released in first-in-first-out order. They are released in response to calls to pthread_cond_signal (which releases the first thread in line) and pthread_cond_broadcast (which releases all threads). However, before a released thread may return from pthread_cond_wait, it first relocks the mutex. Thus only one thread at a time actually returns from pthread_cond_wait. If a call to either routine is made when no threads are queued on the condition variable, nothing happens — the fact that a call had been made is not remembered.

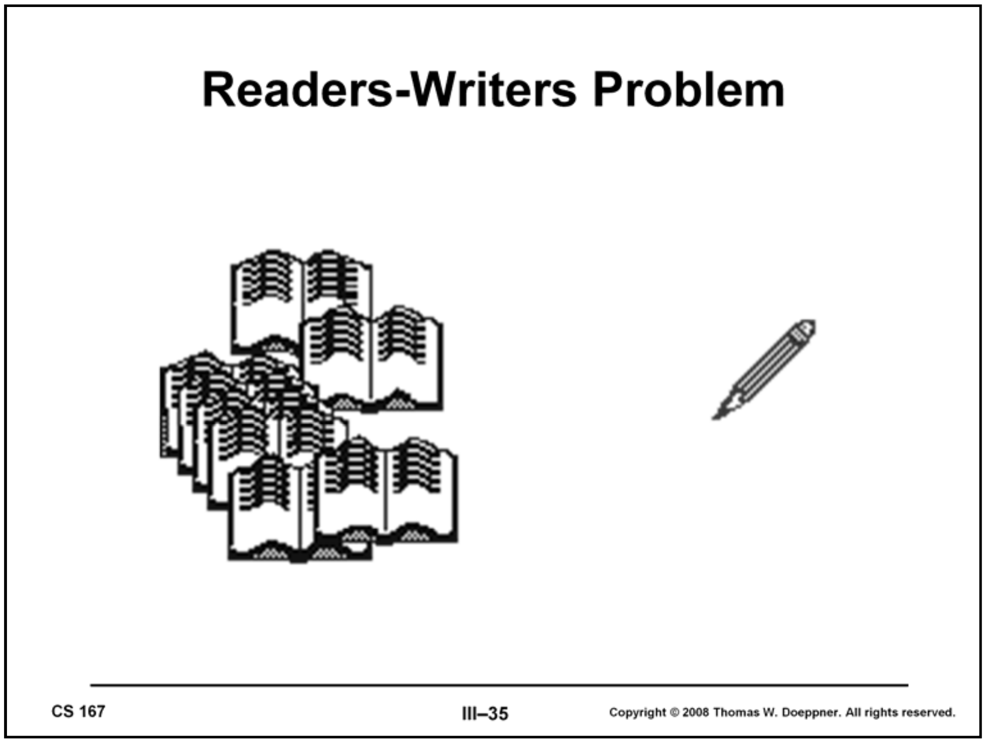
So far, though complicated, the description is rational. Now for the weird part: *a thread may be released from the condition-variable queue at any moment*, perhaps spontaneously, perhaps due to sun spots. Thus it's extremely important that, after *pthread_cond_wait* returns, that the caller check to make sure that it really should have returned. The reason for this weirdness is that it allows a fair amount of latitude in implementations. However, the Linux implementation behaves rationally, i.e., as in the first paragraph.



Set Up

```
int pthread_cond_init(pthread_cond_t *cvp,
    pthread_condattr_t *attrp)

int pthread_cond_destroy(pthread_cond_t *cvp)

int pthread_condattr_init(pthread_condattr_t *attrp)

int pthread_condattr_destroy(pthread_condattr_t *attrp)
```

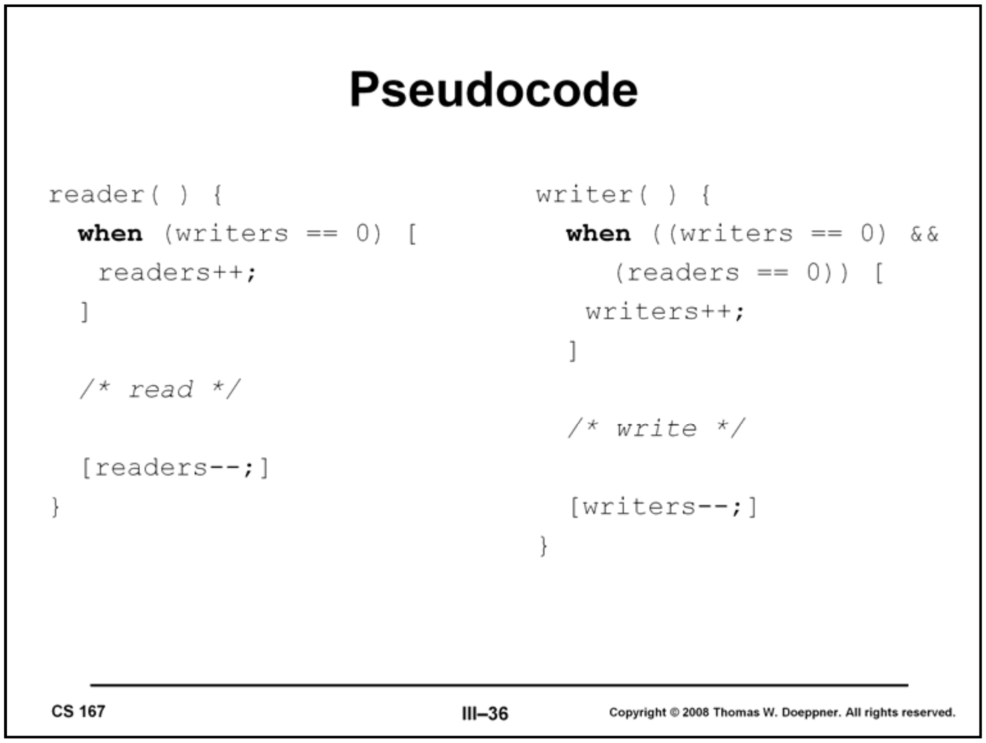
Setting up condition variables is done in a similar fashion as mutexes: The routines *pthread_cond_init* and *pthread_cond_destroy* are supplied to initialize and to destroy a condition variable. They may also be statically initialized by setting them to PTHREAD_COND_INITIALIZER in their declarations. As with mutexes and threads, default attributes may be specified by supplying a zero. The routines *pthread_condattr_init* and *pthread_condattr_destroy* control the initialization and destruction of their attribute structures. However, they are useless on Linux, since no attributes for condition variables are supported.



Readers-Writers Problem

CS 167 III–35 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

Let's look at another standard synchronization problem—the *readers-writers problem*. Here we have some sort of data structure to which any number of threads may have simultaneous access, as long as they are just reading. But if a thread is to write in the data structure, it must have exclusive access.

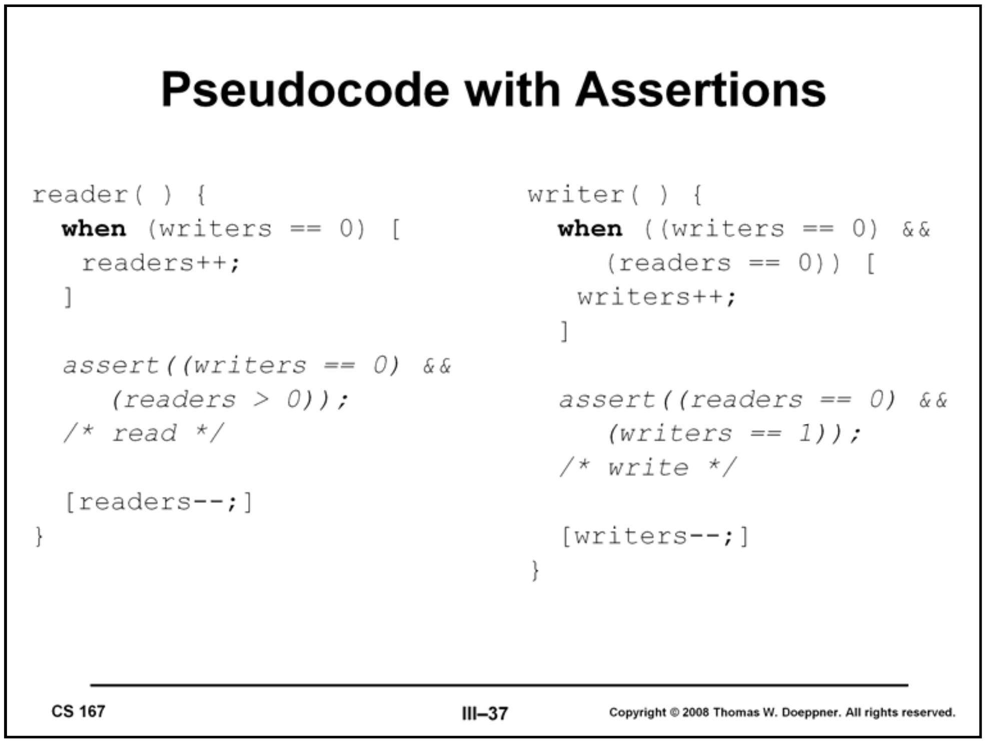


Pseudocode

```
reader( ) {
  when (writers == 0) [
   readers++;
  ]

  /* read */

  [readers--;]
}
```

```
writer( ) {
  when ((writers == 0) &&
     (readers == 0)) [
   writers++;
  ]

  /* write */

  [writers--;]
}
```

Here we again use guarded commands to describe our solution.

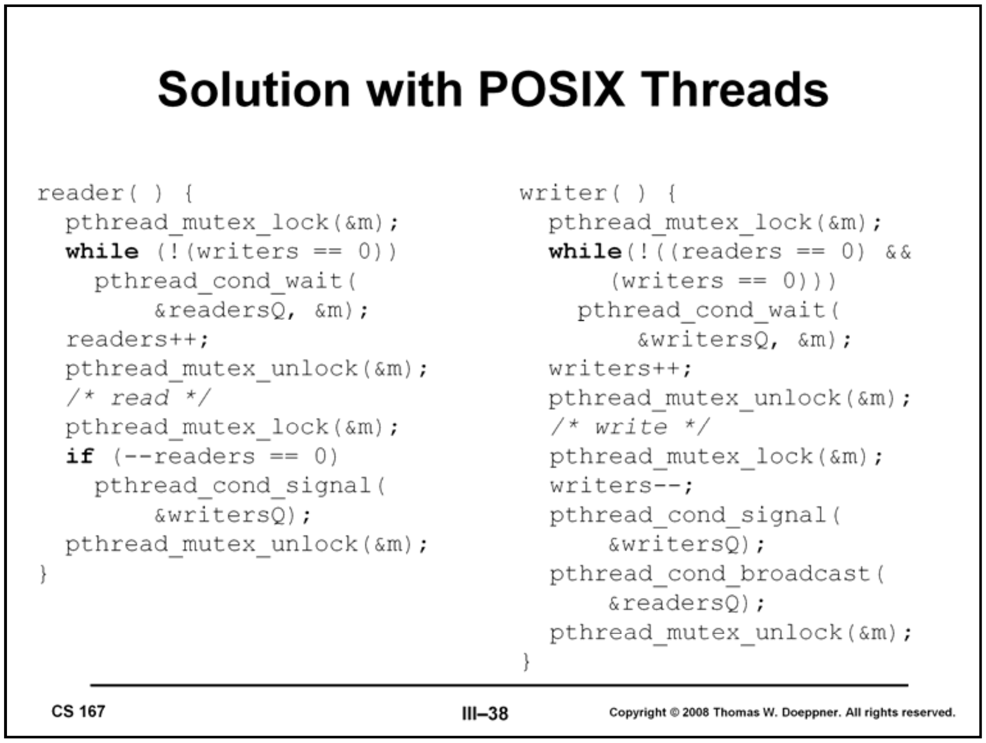


Pseudocode with Assertions

```
reader( ) {                          writer( ) {
  when (writers == 0) [                when ((writers == 0) &&
   readers++;                              (readers == 0)) [
  ]                                     writers++;
                                       ]
  assert((writers == 0) &&
      (readers > 0));                  assert((readers == 0) &&
  /* read */                               (writers == 1));
                                       /* write */
  [readers--;]
}                                      [writers--;]
                                     }
```

CS 167 III–37 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

We've attached assertions to our pseudocode to help make it clearer that our code is correct. The use of assertions is a valuable technique (even in real code), particularly for multithreaded programs.

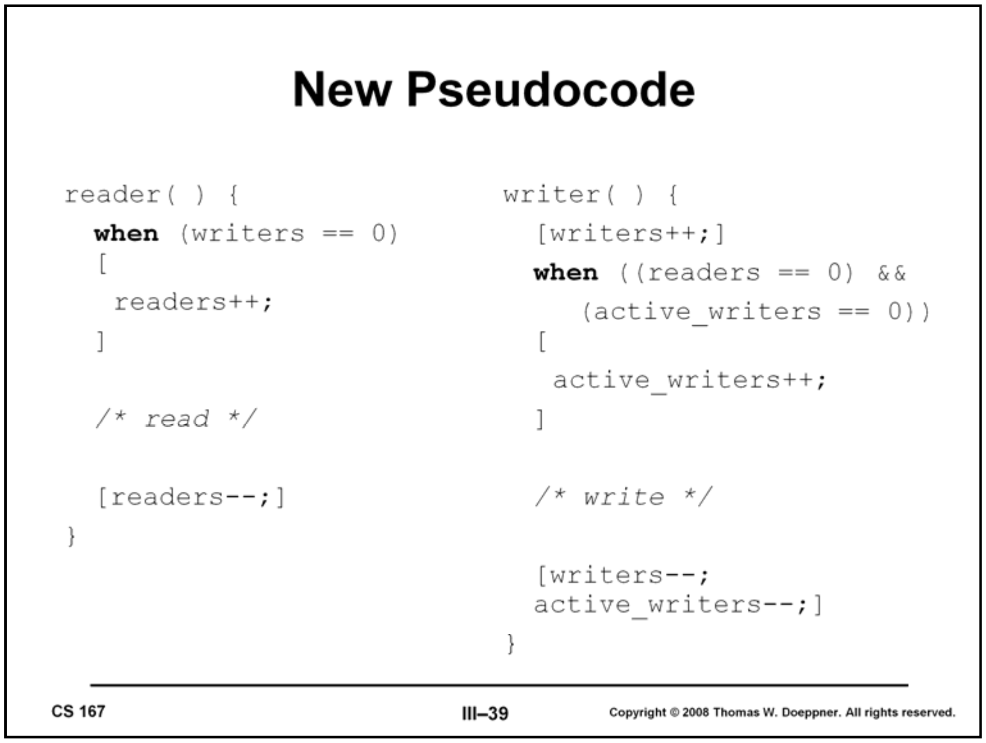


Solution with POSIX Threads

```
reader( ) {
  pthread_mutex_lock(&m);
  while (!(writers == 0))
    pthread_cond_wait(
        &readersQ, &m);
  readers++;
  pthread_mutex_unlock(&m);
  /* read */
  pthread_mutex_lock(&m);
  if (--readers == 0)
    pthread_cond_signal(
        &writersQ);
  pthread_mutex_unlock(&m);
}
```

```
writer( ) {
  pthread_mutex_lock(&m);
  while(!((readers == 0) &&
      (writers == 0)))
    pthread_cond_wait(
        &writersQ, &m);
  writers++;
  pthread_mutex_unlock(&m);
  /* write */
  pthread_mutex_lock(&m);
  writers--;
  pthread_cond_signal(
      &writersQ);
  pthread_cond_broadcast(
      &readersQ);
  pthread_mutex_unlock(&m);
}
```

Now we convert the pseudocode to real code. We use two condition variables, *readersQ* and *writersQ*, to represent queues of readers and writers waiting for notification that their respective guards are true.



New Pseudocode

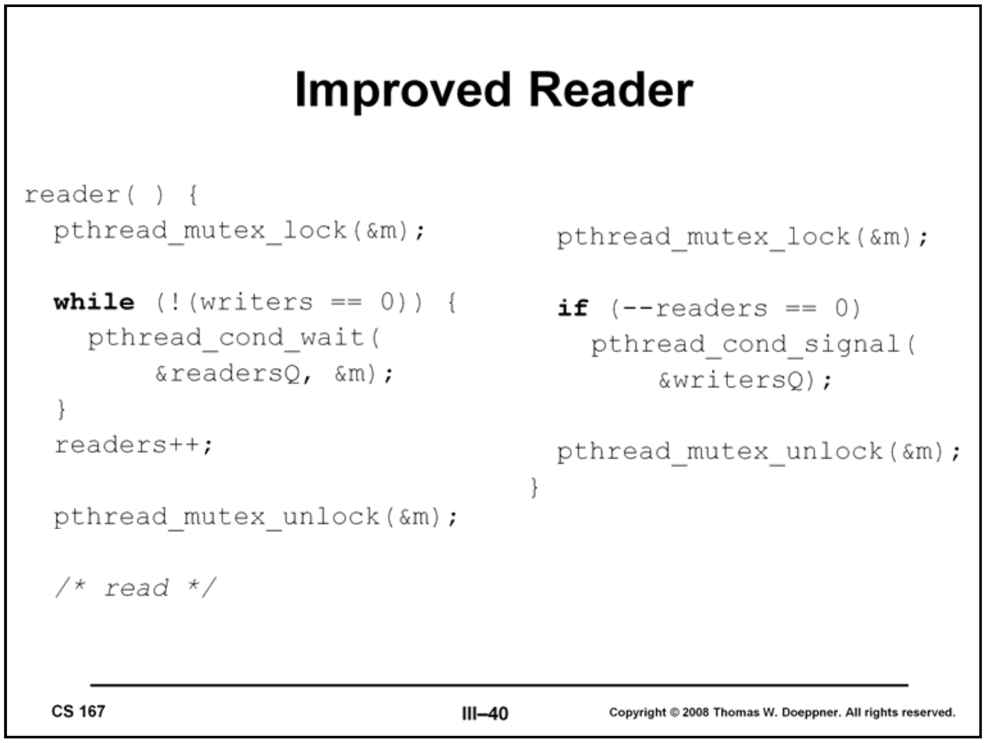
```
reader( ) {
  when (writers == 0)
  [
   readers++;
  ]

  /* read */

  [readers--;]
}
```

```
writer( ) {
  [writers++;]
  when ((readers == 0) &&
     (active_writers == 0))
  [
   active_writers++;
  ]

  /* write */

  [writers--;
  active_writers--;]
}
```

It turns out that our solution to the readers-writers problem has a flaw: writers may have to wait indefinitely before being allowed to write. This is because as long as there is a reader reading, further readers are allowed in and writers are prevented from writing.

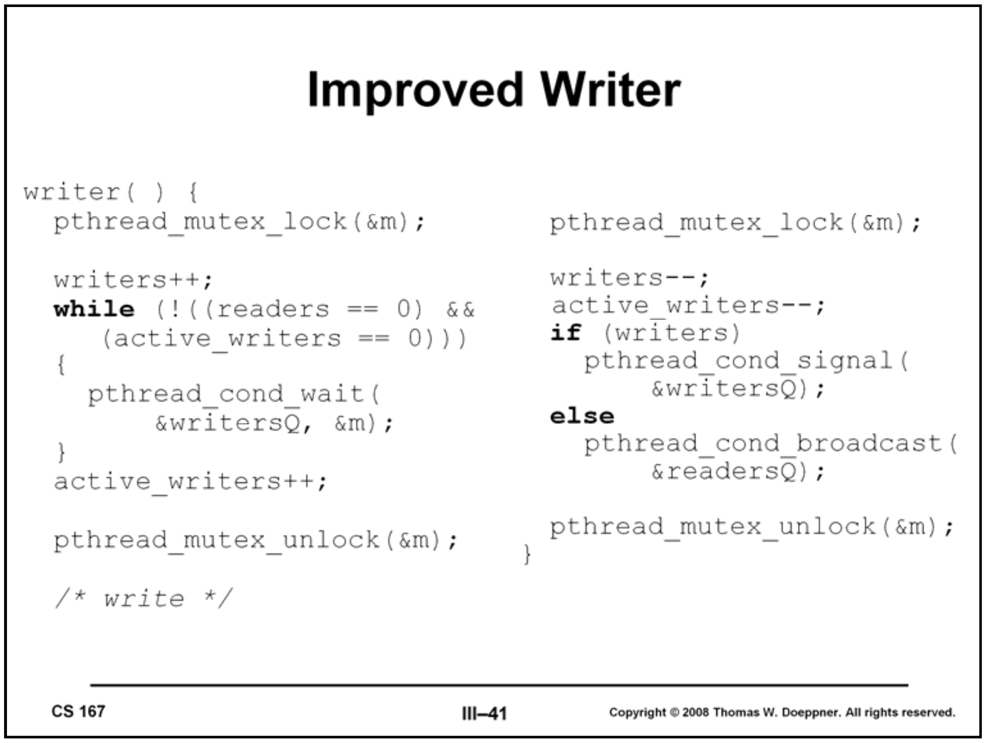
Though one might argue that the best solution is one that is fair to both readers and writers, what is usually preferred is one that favors writers—i.e., readers requesting permission to read must yield to writers, but writers do not yield to readers.

This slide gives pseudocode using guarded commands for a new solution to the problem, a writers-priority solution. Writers indicate their intention to write by incrementing *writers*. We use the variable *active_writers* to indicate how many writers are currently writing.


Improved Reader

```
reader( ) {
  pthread_mutex_lock(&m);

  while (!(writers == 0)) {
    pthread_cond_wait(
        &readersQ, &m);
  }
  readers++;

  pthread_mutex_unlock(&m);

  /* read */

  pthread_mutex_lock(&m);

  if (--readers == 0)
    pthread_cond_signal(
        &writersQ);

  pthread_mutex_unlock(&m);
}
```

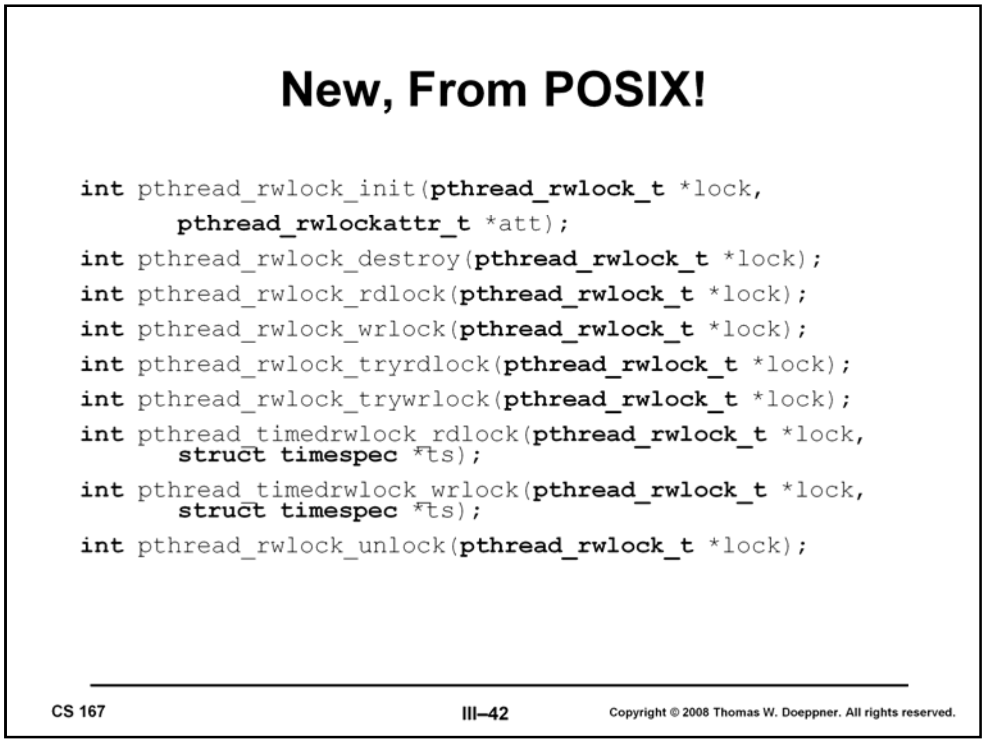
In this slide we've taken the pseudocode for the writers-priority reader and translated it into legal POSIX.


Improved Writer

```
writer( ) {
  pthread_mutex_lock(&m);

  writers++;
  while (!((readers == 0) &&
     (active_writers == 0)))
  {
    pthread_cond_wait(
        &writersQ, &m);
  }
  active_writers++;

  pthread_mutex_unlock(&m);

  /* write */

  pthread_mutex_lock(&m);

  writers--;
  active_writers--;
  if (writers)
    pthread_cond_signal(
        &writersQ);
  else
    pthread_cond_broadcast(
        &readersQ);

  pthread_mutex_unlock(&m);
}
```

CS 167 III–41 Copyright © 2008 Thomas W. Doeppner. All rights reserved.

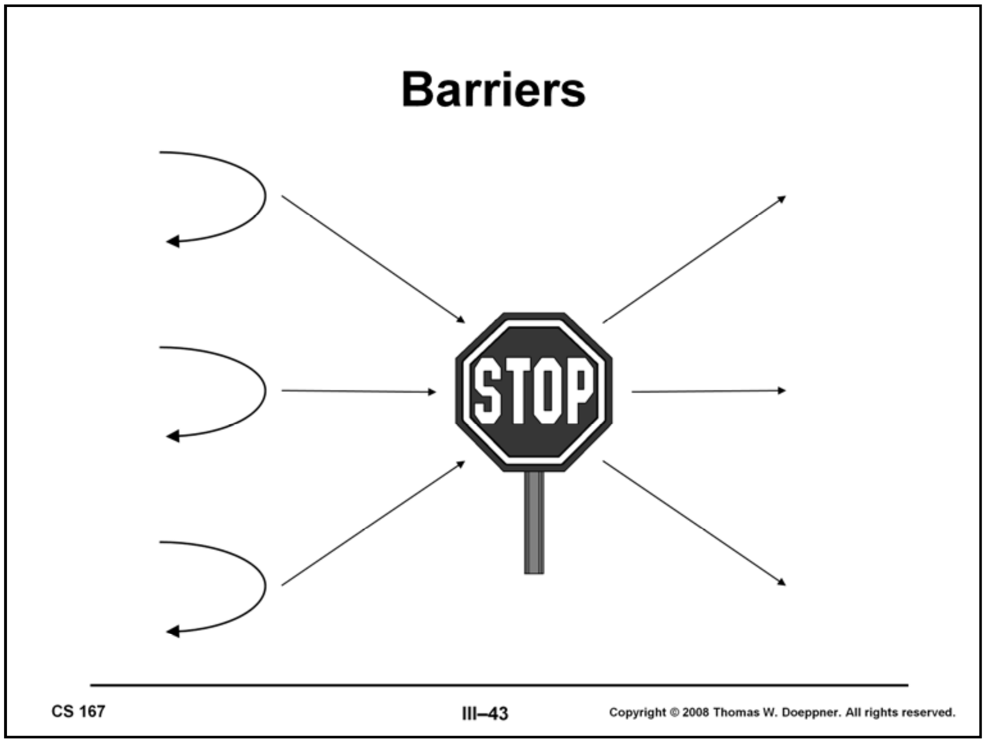
Here's the POSIX version of the writer code. Note the use of *pthread_cond_broadcast*: we use it to insure that all currently waiting readers are released.



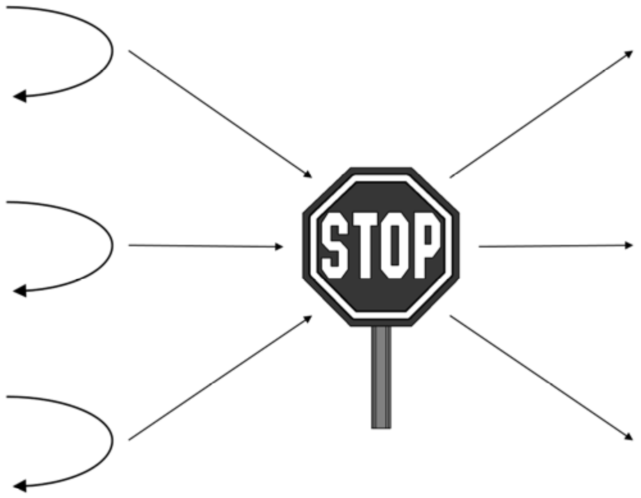
New, From POSIX!

```
int pthread_rwlock_init(pthread_rwlock_t *lock,
        pthread_rwlockattr_t *att);
int pthread_rwlock_destroy(pthread_rwlock_t *lock);
int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
int pthread_timedrwlock_rdlock(pthread_rwlock_t *lock,
        struct timespec *ts);
int pthread_timedrwlock_wrlock(pthread_rwlock_t *lock,
        struct timespec *ts);
int pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

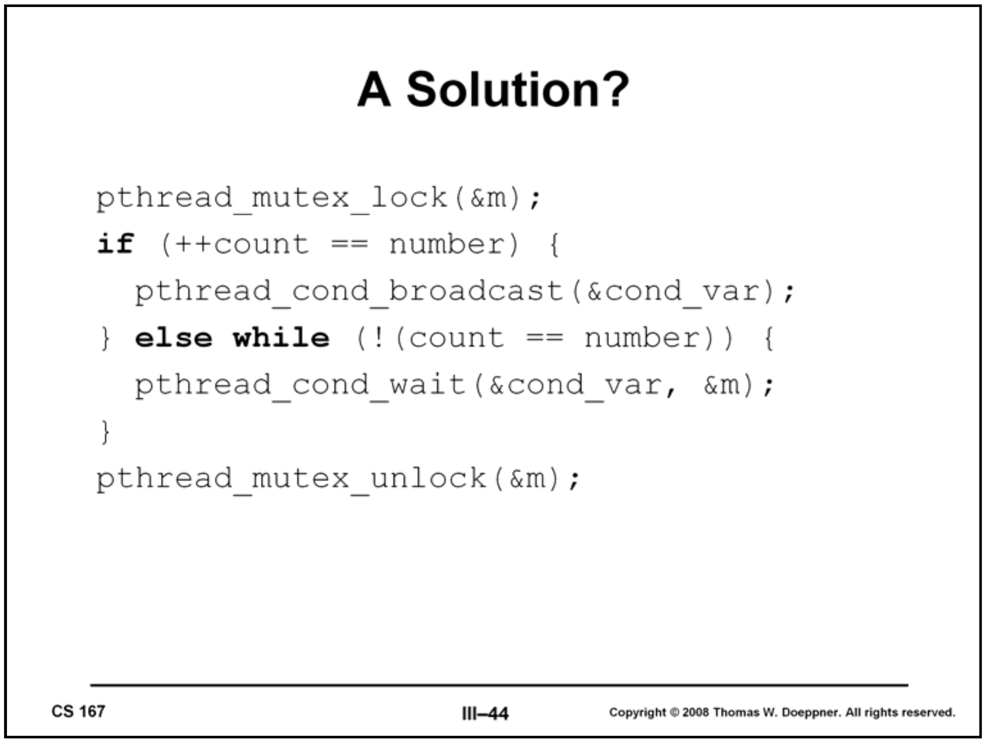
With POSIX 1003.1j support for readers-writers locks was finally introduced. The almost complete API is shown in the slide (what's missing are the operations on attributes). As might be expected, readers-writers locks can be statically initialized with the constant PTHREAD_RWLOCK_INITIALIZER. The "timedrwlock" routines allow one to wait until the lock is available or a time-limit is exceeded, whichever comes first.



Barriers



A *barrier* is a conceptually simple and very useful synchronization construct that, unfortunately, is not part of the POSIX-threads API (nor of the Win32-threads API). A barrier is established for some predetermined number of threads; threads call the barrier's *wait* routine to enter it; no thread may exit the barrier until all threads have entered it.

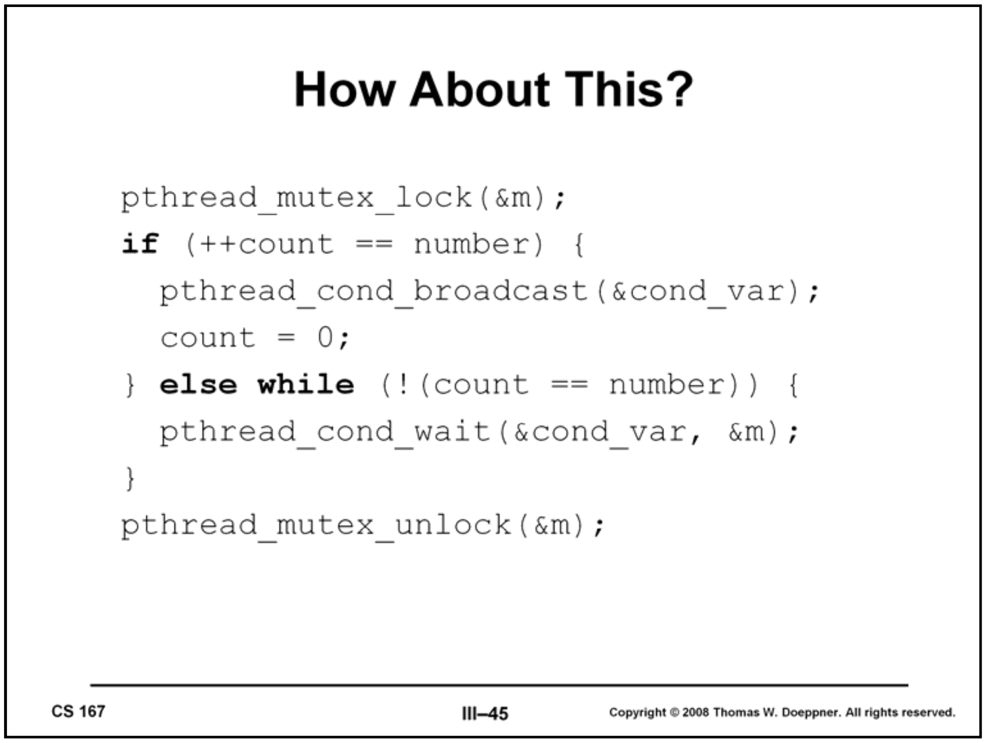


A Solution?

```
pthread_mutex_lock(&m);
if (++count == number) {
  pthread_cond_broadcast(&cond_var);
} else while (!(count == number)) {
  pthread_cond_wait(&cond_var, &m);
}
pthread_mutex_unlock(&m);
```

CS 167 III–44

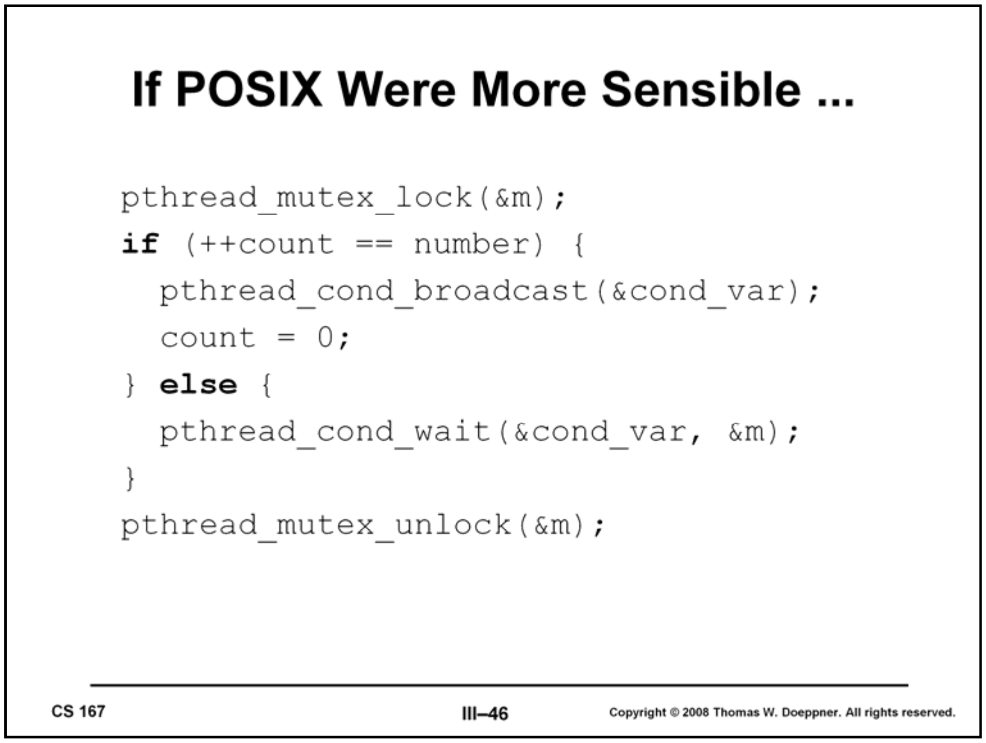
Is this a correct solution?



How About This?

```
pthread_mutex_lock(&m);
if (++count == number) {
  pthread_cond_broadcast(&cond_var);
  count = 0;
} else while (!(count == number)) {
  pthread_cond_wait(&cond_var, &m);
}
pthread_mutex_unlock(&m);
```

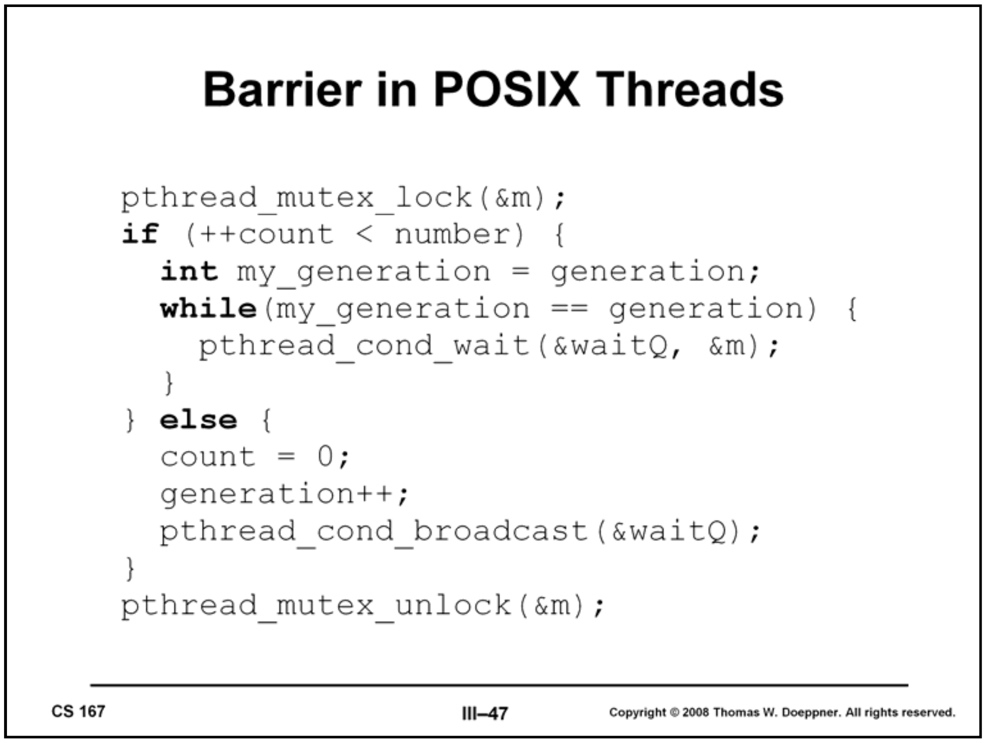
How about this?



If POSIX Were More Sensible ...

```
pthread_mutex_lock(&m);
if (++count == number) {
  pthread_cond_broadcast(&cond_var);
  count = 0;
} else {
  pthread_cond_wait(&cond_var, &m);
}
pthread_mutex_unlock(&m);
```

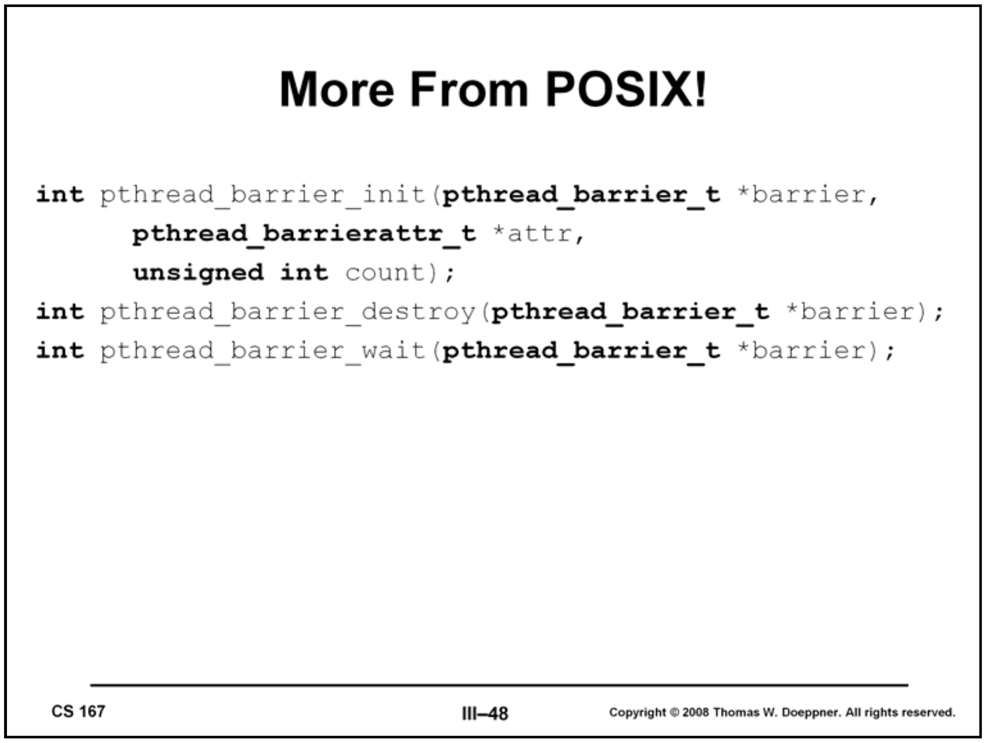
If *pthread_cond_wait* had sane semantics (i.e., if threads were released only in response to calls to *pthread_cond_signal* and *pthread_cond_broadcast*), this would work.



Barrier in POSIX Threads

```
pthread_mutex_lock(&m);
if (++count < number) {
  int my_generation = generation;
  while(my_generation == generation) {
    pthread_cond_wait(&waitQ, &m);
  }
} else {
  count = 0;
  generation++;
  pthread_cond_broadcast(&waitQ);
}
pthread_mutex_unlock(&m);
```

Implementing barriers in POSIX threads is not trivial. Since count, the number of threads that have entered the barrier, will be reset to 0 once all threads have entered, we can't use it in the guard. But, nevertheless, we still must wakeup all waiting threads as soon as the last one enters the barrier. We accomplish this with the *generation* global variable and the *my_generation* local variable. An entering thread increments count and joins the condition-variable queue if it's still less than the target number of threads. However, before it joins the queue, it copies the current value of *generation* into its local *my_generation* and then joins the queue of waiting threads, via *pthread_cond_wait*, until *my_generation* is no longer equal to *generation*. When the last thread enters the barrier, it increments generation and wakes up all waiting threads. Each of these sees that its private *my_generation* is no longer equal to *generation*, and thus the last thread must have entered the barrier.



More From POSIX!

```
int pthread_barrier_init(pthread_barrier_t *barrier,
      pthread_barrierattr_t *attr,
      unsigned int count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

As part of POSIX 1003.1j, barriers were introduced. Unlike other POSIX-threads objects, they cannot be statically initialized; one must call *pthread_barrier_init* and specify the number of threads that must enter the barrier. In some applications it might be necessary for one thread to be designated to perform some sort function on behalf of all of them when all exit the barrier. Thus *pthread_barrier_wait* returns PTHREAD_BARRIER_SERIAL_THREAD in one thread and zero in the others on success.